DOCTEUR EN SCIENCES

Application des Machines à Etats Finis en Synthèse de la Parole
Sélection d'unités non uniformes et Correction orthographique

Beaufort, Richard

Award date:
2008

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix – Namur

Faculté d'Informatique

Application des Machines à Etats Finis en Synthèse de la Parole

Sélection d'unités non uniformes et Correction orthographique

Dissertation présentée par

Richard Beaufort

en vue de l'obtention du grade de

Docteur en Sciences, orientation Informatique

4 mars 2008

Composition du jury

Prof. Guy Deville, ELV, FUNDP, Namur	co-promoteur
Prof. Thierry Dutoit, TCTS, FPMs, Mons	co-promoteur
Prof. Christian Fluhr, INSTN, CEA, Saclay, France	examinateur
Prof. Jean-Marie Jacquet, Faculté d'Informatique, FUNDP, Namur	promoteur
Prof. Pierre-Yves Schobbens, Faculté d'Informatique, FUNDP, Namur	examinateur
Prof. Wim Vanhoof, Faculté d'Informatique, FUNDP, Namur	président du jury

Application des Machines à Etats Finis en Synthèse de la Parole

Sélection d'unités non uniformes et Correction orthographique

Richard Beaufort

Jean-Marie Jacquet (*promoteur*) Guy Deville (*co-promoteur*) Thierry Dutoit (*co-promoteur*)

Résumé

Les machines à états finis sont des outils puissants : équivalents informatiques des langages réguliers et des relations régulières, elles s'accompagnent d'algorithmes d'optimisation et peuvent être exprimées sous la forme d'expressions régulières et de règles de réécriture. Présentes depuis l'apparition de l'informatique et abondamment employées dans les domaines les plus variés, les machines à états finis ont cependant été délaissées en traitement automatique de la langue, domaine où les chercheurs ont préféré recourir à des outils plus adaptés à la description syntaxique.

La recherche a cependant récemment réalisé un véritable bond en avant dans le domaine des machines à états finis pondérées, proposant de nouveaux algorithmes qui ont ouvert la voie à une nouvelle technologie du langage, capable de modéliser ce degré d'incertitude qui est indispensable à certains domaines du traitement automatique de la langue.

La synthèse de la parole est l'un de ces domaines. Le processus de synthèse de la parole, qui consiste à produire de la parole à partir du texte, regorge de difficultés, empreintes d'un certain degré d'incertitude, qui peuvent utilement profiter du potentiel expressif des outils à états finis dans leur ensemble.

Dans le cadre de cette thèse, deux de ces difficultés ont particulièrement retenu notre attention : l'étape de sélection d'unités non uniformes qui précède la génération de la parole, et la correction orthographique réalisée dans le contexte de l'analyse morpho-syntaxique.

Les travaux réalisés s'articulent autour de trois objectifs distincts. D'une part, nous avons désiré prouver que les machines à états finis facilitent la conception de tâches complexes sous la forme d'une succession de tâches simples. D'autre part, nous avons voulu proposer des approches nouvelles et originales en sélection d'unités non uniformes et en correction orthographique. Enfin, nous avons tâché de contribuer au domaine des machines à états finis, en définissant de nouvelles méthodes d'implémentation et de nouveaux algorithmes qui ont été intégrés dans nos propres outils : une bibliothèque de machines à états finis, et un compilateur de dictionnaires, de langages et de règles de réécriture.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des acronymes

AUF	**A**gence **U**niversitaire de la **F**rancophonie
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange *code américain standard pour l'échange d'informations*
DFA	**D**eterministic **F**inite-State **A**utomaton *Automate à états finis déterministe*
DLS	**D**ata **L**ayer **S**tructure *Structure de données en niveaux*
eLite	**E**nhanced, **Li**nguistically-based **Te**xt-to-speech (synthesis system) *(Système de synthèse) de la parole à partir du texte,* *amélioré et basé sur de la linguistique*
FSA	**F**inite-**S**tate **A**utomaton *Automate à états finis*
FSM	**F**inite-**S**tate **M**achine *Machine à états finis*
FST	**F**inite-**S**tate **T**ransducer *Transducteur à états finis*
IP	**I**nternet **P**rotocol *protocole Internet*
IV	**I**n-**V**ocabulary (word) *(Mot) appartenant au dictionnaire*
KHz	**K**ilo **H**ertz
HMM	**H**idden **M**arkov **M**odel *Modèle de Markov caché*
ICDAR	**I**nternational **C**onference on **D**ocument **A**nalysis and **R**ecognition *Conférence internationale sur l'analyse et la reconnaissance de documents*
LiONS	**Li**nguistically **O**riented **N**on-uniform units **S**election (system) *(Système de) sélection d'unités non uniformes orienté linguistique*
LSA	**L**atent **S**emantic **A**nalysis *Analyse sémantique latente*
LSI	**L**atent **S**emantic **I**ndexing *Indexation sémantique latente*

MLDS	**M**ulti **L**ayers **D**ata **S**tructure *Structure de données multi-niveaux*
NFA	**N**on-deterministic **F**inite-State **A**utomaton *Automate à états finis non déterministe*
NLP	**N**atural **L**anguage **P**rocessing *Traitement du langage naturel*
OCR	**O**ptical **C**haracter **R**ecognition *Reconnaissance optique de caractères*
OOV	**O**ut-**O**f-**V**ocabulary (word) *(Mot) hors vocabulaire*
PDA	**P**ersonal **D**igital **A**ssitant *Agenda électronique*
SDRAM	**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory *Mémoire Dynamique Synchrone*
SMS	**S**hort **M**essage **S**ervice *Service de messages courts*
SVD	**S**ingular **V**alue **D**ecomposition *Décomposition en valeurs singulières*
SVM	**S**upport **V**ector **M**achine *Machine à support vectoriel*
TAL(N)	**T**raitement **A**utomatique du **L**angage (**N**aturel)
TTS	**T**ext-**T**o-**S**peech (synthesis) *(Synthèse) de la parole à partir du texte*
URI	**U**Uniform **R**esource **I**dentifier *Identificateur uniforme de ressource*
WFSA	**W**eighted FSA *Automate à états finis pondéré*
WFSM	**W**eighted FSM *Machine à états finis pondérée*
WFST	**W**eighted FST *Transducteur à états finis pondéré*
WNFA	**W**eighted NFA *Automate à états finis pondéré non déterministe*
ϵ-NFA	NFA with *epsilon* (ϵ) transitions *Automate à états finis non déterministe pourvu de transitions ϵ*

Comme beaucoup d'expériences de vie, une thèse est certainement le fruit du hasard et le résultat de multiples rencontres. Linguiste de formation, je me suis intéressé à l'informatique sur le tard, alors que je m'apprêtais à me diriger vers l'enseignement. Un jour où je flânais dans le hall de la Faculté de Philosophie & Lettres à Louvain-la-Neuve, je me suis arrêté devant une affiche qui vantait les mérites de l'ingénierie linguistique. J'ai lu, et je me suis dit : « Pourquoi pas ? » J'ai alors complété ma formation par deux années passionnantes, qui m'ont permis de suivre des cours dans quatre universités wallonnes, avant de partir en France pour une année très enrichissante.

Si j'ai eu la chance de commencer cette thèse, je le dois entièrement à Thierry Dutoit, que j'avais eu comme professeur, et qui m'a proposé de revenir de France pour m'intéresser aux machines à états finis. Il m'a dit : « Tu verras, le domaine est en pleine ébullition ! » Je pense aujourd'hui qu'il avait raison. Je le remercie de m'avoir fait cette proposition, qui m'a donné accès à un domaine incroyablement intéressant. Mais je le remercie encore plus de m'avoir fait confiance : je ne connaissais rien du domaine lorsque j'ai commencé, et il y avait tant à apprendre ! Thierry y a cru, pour nous deux. Cela m'a donné l'occasion de le côtoyer pendant plusieurs années. Le rêve : j'ai travaillé avec Thierry Dutoit, l'un des chercheurs les plus réputés en synthèse de la parole !

J'ai été accueilli au sein de l'Institut d'informatique de Namur par Guy Deville, dont j'avais également suivi les cours, et qui a proposé de me prendre en thèse. Sans lui, ce projet ne serait certainement pas devenu réalité. Je tiens à lui adresser toute ma reconnaissance pour cela.

J'ai ensuite été orienté vers Jean-Marie Jacquet, mon promoteur, qui a accepté de me faire confiance, a priori. C'est une chance que l'on rencontre rarement, je pense. J'ai reçu de lui des conseils précieux et une écoute attentive. Lors de la rédaction de ce document, il a en outre réalisé un travail de relecture d'une qualité que je tiens à souligner, tant cela m'a aidé à avancer et à conclure. Je le remercie chaleureusement de m'avoir accordé autant de temps.

Dans le cadre de projets de recherche, j'ai rencontré Cédrick Fairon, qui dirige le CENTAL à Louvain-la-Neuve. De discussions en réunions, j'ai appris à apprécier son dynamisme naturel et ses qualités humaines. Comme le dit l'une de ses plus grandes amies,

Cédrick a le don de trouver le mot juste : c'est lui qui a su m'inciter à entamer la rédaction de cette thèse. Qu'a-t-il dit exactement ? Je ne sais plus. Mais il est certain que je suis sorti de son bureau, convaincu qu'il y avait matière à entamer la rédaction.

La thèse a été réalisée au sein du centre de recherche Multitel ASBL, situé à Mons. Si cela a été possible, c'est grâce aux directeurs du Centre, messieurs Dominique Derestiat et Jean-Christophe Froidure, qui ont accepté que je consacre une partie de mon temps à l'avancement de mes recherches. Cette dernière année, ce sont eux qui ont accepté que je me concentre sur la rédaction. C'est une faveur énorme qu'ils m'ont accordée, et pour laquelle je tiens à leur témoigner toute ma gratitude.

Au moment où j'écris ces lignes, la rédaction est terminée. Je m'apprête à faire imprimer ce document et à l'envoyer à mes lecteurs. Je voudrais profiter de cet instant pour signifier à Christian Fluhr, Pierre-Yves Schobbens et Wim Vanhoof, l'immense honneur qu'ils me font en acceptant de participer à mon jury.

Je retiendrai tout particulièrement de cette thèse qu'elle aura été l'occasion pour moi de m'enrichir au contact d'autres chercheurs. J'ai entre autres eu la joie de collaborer avec Vincent Colotte, du LORIA [1]. Ensemble, nous avons mis au point la première version du système de sélection LiONS. J'ai également eu l'opportunité de travailler avec Céline Thillou, de la Faculté Polytechnique de Mons. L'objectif était d'intégrer un module de correction orthographique dans son système de reconnaissance des caractères en scènes naturelles. Dans les deux cas, ce fut un véritable plaisir, tant sur le plan scientifique que sur le plan humain.

Si ce travail de thèse a abouti, c'est aussi parce qu'il a été réalisé au sein de l'équipe TTS de Multitel. C'est ainsi que j'ai eu la joie et le privilège de travailler, jour après jour, avec Alain Ruelle et Sophie Roekhaut. Alain est certainement l'informaticien le plus doué qu'il m'ait été donné de rencontrer : à son contact, j'ai personnellement énormément appris en informatique. Sophie est une chercheuse dont la vivacité m'a toujours émerveillé : elle a le don de dégrossir les problèmes, trouvant des solutions là où d'autres resteraient perplexes. Tous les deux m'ont énormément apporté sur le plan humain. Je sais que je peux compter sur eux en toute circonstance. J'espère que, quels que soient les aléas de la vie, nous resterons en contact...

Je tiens également à témoigner toute la gratitude et toute l'affection que j'éprouve pour ma famille et mes amis proches.

Mes premiers remerciements vont tout naturellement à mes parents, sans qui tout ceci n'aurait tout simplement pas été. Je pense ne pas me tromper en disant qu'ils ont été mes premiers « fans » : ils ont cru en moi, et m'ont poussé à me dépasser, entre autres au travers de cette thèse. Je remercie chaleureusement Maman, qui m'a soutenu moralement en me mijotant de si bons petits plats, dont elle a le secret ! Et je remercie tout particulièrement

[1] Laboratoire d'informatique de Lorraine, Nancy, France.

Papa, qui a été le seul, en dehors des membres de mon jury, à relire la totalité de ce document. La minutie dont il a fait montre est impressionnante ! Papa m'a également servi d'exemple, un exemple que j'ai tâché de suivre tout au long de la rédaction : je me vois encore petit, en train d'écouter ou de lire l'une ou l'autre lettre qu'il avait rédigée. A chaque fois, j'étais frappé par la clarté de la structure et, de ce fait, par la facilité à saisir les idées, quelles qu'elles fussent.

Cette dernière année, j'ai pu énormément compter sur le soutien de mes beaux-parents, qui ont régulièrement gardé ma fille, Héloïse. Leur aide précieuse m'a permis de rédiger l'esprit tranquille, et je tiens à leur exprimer toute ma reconnaissance pour cela.

Je voudrais également remercier mes deux médecins préférés, mon oncle Michel Baudoux et une amie très chère, Emmanuelle Jadoul, qui m'ont soutenu psychologiquement aux moments où j'en avais le plus besoin. Sans les discussions que j'ai eues avec eux, et les conseils ponctuels qu'ils m'ont donnés, je ne suis pas sûr que j'aurais eu la force morale de continuer.

Je tiens aussi à saluer mes chers amis, Philippe Debauche, Eric Brasseur, Chantal Mairiaux, ainsi que les épéistes de La Licorne et leurs familles, qui m'ont incontestablement aidé par le regard amical qu'ils ont porté sur mes agissements parfois étranges.

Enfin, je me tourne vers celle qui illumine ma vie depuis maintenant 11 ans. Anne-Sophie, je ne sais comment te dire merci. Je dois probablement commencer par te demander pardon. Pardon d'avoir été aussi peu présent ces derniers temps. J'ai été accaparé de longs mois par une rédaction dont je ne voyais pas la fin, et dont tu subissais tous les désagréments, de mon absence fréquente à mes brusques changements d'humeur. Je t'ai laissé la totalité de notre ménage, et l'éducation de notre fille, en surplus de ton travail qui te demande tant de temps. Et pourtant, tu t'es si bien acquittée de ce fardeau indescriptible ! Je t'en remercie profondément, amoureusement, et j'espère pouvoir t'en donner la preuve tangible. Je peux en tout cas te promettre une chose : je suis à nouveau là, pour toi, pour notre fille, pour nous.

Richard Beaufort

A ma famille...

Introduction

1 Du contexte aux objectifs

Le travail de thèse que nous présentons dans ce document a été réalisé au sein de l'équipe de synthèse de la parole du Centre de recherche *Multitel ASBL* [2]. Ce travail s'est résolument inscrit dans les perspectives de recherche du groupe. C'est ainsi que la thèse s'est orientée vers l'application des machines à états finis en synthèse de la parole.

Les machines à états finis (FSMs [3]) qui interviennent classiquement en traitement du langage naturel sont les automates et les transducteurs. Présentes depuis l'apparition de l'informatique, ces machines ont également été abondamment employées dans des domaines aussi variés que la compilation de programme, la modélisation de circuits et la gestion de bases de données.

Les automates et les transducteurs permettent de représenter respectivement des langages et des relations entre langages, sous la forme de graphes orientés étiquetés. L'intérêt de cette représentation graphique est qu'elle s'accompagne d'un ensemble d'algorithmes d'optimisation : les graphes peuvent être déterminisés et minimisés, de sorte que le langage ou la relation soit représenté en un minimum de place, et que le parcours d'une entrée à analyser ne dépende pas de la taille du graphe, mais soit linéairement proportionnel à la taille de l'entrée. Ces machines sont en outre équivalentes aux langages réguliers. Cette équivalence est de première importance, parce qu'elle autorise l'utilisation des opérations régulières sur les FSMs, telles que l'union, la concaténation ou l'étoile de Kleene. Les transducteurs, quant à eux, acceptent entre autres également l'opération de composition, opération majeure parce qu'elle permet de modéliser des relations complexes à partir de relations beaucoup plus simples.

Modéliser un langage ou une relation complexe, directement sous la forme d'un graphe orienté étiqueté, est cependant une tâche ardue et risquée : il n'est pas assuré que le graphe construit représente effectivement le langage désiré. Heureusement, l'équivalence entre FSMs et langages réguliers permet de recourir à des modes d'expression bien établis, dont l'équivalence avec les langages réguliers a également été démontrée : les expressions

[2] Le Centre se situe à Mons, Belgique.
[3] *Finite-State Machines.*

régulières et les règles de réécriture, à partir desquelles il est possible de construire des machines équivalentes. Ensemble, ces descriptions syntaxiques et les machines à états finis forment une « boîte à outils » certainement utile au traitement du langage naturel.

Cependant, l'intérêt des chercheurs pour ces outils a été remis en cause dans le domaine du traitement du langage naturel, parce que les langages réguliers n'autorisent pas d'imbrications infinies, base selon Chomsky de toute grammaire digne de ce nom. L'arsenal algorithmique lié aux FSMs a dès lors été abandonné en traitement du langage naturel, au profit de formalismes jugés plus puissants, comme les grammaires hors-contexte.

Il est vrai que la puissance des automates et des transducteurs ne suffit pas à certains domaines du traitement automatique des langues, qui ont besoin d'outils capables de gérer un certain *degré d'incertitude*. Or, des chercheurs ont récemment proposé des extensions aux algorithmes classiques définis sur les automates et les transducteurs, applicables aux automates et transducteurs pondérés. De nouveaux algorithmes ont également été modélisés, comme la recherche des n meilleurs chemins d'un graphe pondéré. Ces extensions ont dès lors ouvert la voie à une nouvelle technologie du langage basée sur des machines à états finis pondérées, là où auparavant *modèles de langue* rimaient inéluctablement avec *programmation dynamique* ou *modèles de Markov cachés*. Ce véritable bond en avant de la recherche a ouvert la voie à une nouvelle technologie du langage, basée sur des machines pondérées et capable de modéliser cette incertitude, indispensable à certaines applications en traitement du langage naturel.

La synthèse de la parole est certainement l'une de ces applications. Le processus de synthèse de la parole consiste à produire de la parole à partir du texte. Il s'agit donc de générer un *signal acoustique* de parole à partir d'une *représentation symbolique*, le texte. Cependant, parce que la langue écrite est hautement ambiguë, un système de synthèse à partir du texte est classiquement divisé en deux modules distincts :

1. Un module de traitement automatique de la langue écrite, qui produit une représentation symbolique *non ambiguë* du texte. Généralement, cette représentation symbolique est constituée de phonèmes et d'informations prosodiques.
2. Un module de traitement du signal, qui produit le signal de parole à partir de cette représentation non ambiguë.

La Figure 1 présente l'architecture d'un système de synthèse classique.

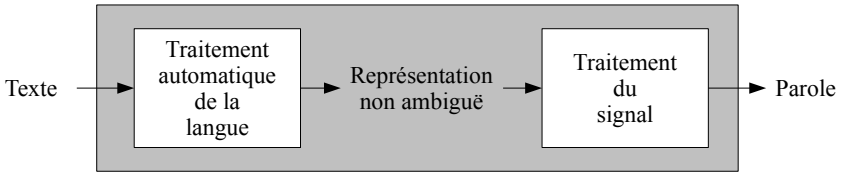


FIG. 1: Système de synthèse de la parole à partir du texte

Les deux modules du système de synthèse regorgent de difficultés, plus ou moins empreintes d'un certain degré d'incertitude. Dans le cadre de cette thèse, deux de ces difficultés ont particulièrement retenu notre attention, parce qu'elle nous semblent pouvoir profiter du potentiel expressif des machines à états finis :

1. *Dans le module de traitement du signal.* L'un des derniers états de l'art dans le domaine est la synthèse par sélection d'unités non uniformes. Ce type de synthèse, au lieu de générer le signal de parole, préfère le reconstituer en concaténant bout à bout des unités de parole extraites d'un corpus dédié. L'objectif de la démarche est de limiter le traitement du signal au strict minimum, afin de conserver le naturel de la voix. Ce type de synthèse repose cependant sur une étape délicate de sélection de la meilleure suite d'unités : il n'est pas évident de concevoir un système de sélection capable de combiner efficacement les bons critères de sélection et la bonne méthode d'optimisation.

   La première application des machines à états finis concerne donc l'étape de ***sélection d'unités non uniformes****.*

2. *Dans le module de traitement de la langue.* La génération de la représentation non ambiguë repose sur une analyse linguistique préalable, dont le rôle est de désambiguïser le texte. Dans l'ensemble, le module dépend fortement de la qualité du texte : si le texte est corrompu et que l'analyse linguistique échoue, la représentation symbolique générée sera erronée. Ce faisant, c'est tout le processus de synthèse qui est mis en danger : dans le flux de parole, certaines erreurs non corrigées seront audibles, soit au niveau phonétique, soit au niveau prosodique.

   La seconde application des machines à états finis que nous proposons concerne de ce fait l'analyse morpho-syntaxique dans son ensemble, et plus particulièrement la ***correction orthographique****.*

2 Plan de la thèse

Partie I. La première partie est consacrée aux machines à états finis. Nous y présentons les notions fondamentales et les algorithmes principaux qui s'y rapportent, avant de décrire les outils que nous avons conçus dans le cadre de cette thèse : une bibliothèque de machines à états finis, et un compilateur d'expressions régulières et de règles de réécriture pondérées. Ces outils intègrent quelques contributions personnelles au domaine.

Partie II. Dans la seconde partie, nous abordons la question délicate de la sélection d'unités non uniformes. Après avoir présenté les concepts sur lesquels repose la synthèse par sélection, nous dressons un état de l'art du domaine. Sur cette base, nous présentons l'approche développée, conçue exclusivement à l'aide de machines à états finis.

Partie III. Cette troisième et dernière partie s'intéresse à la correction orthographique. Deux applications sont proposées. La première, la plus importante, concerne la correction des textes entrés au clavier, et s'intègre à l'analyse morpho-syntaxique du système

de synthèse. La seconde s'inscrit dans le domaine particulier de la reconnaissance de caractères en scènes naturelles. Ces deux applications sont exclusivement réalisées à l'aide de machines à états finis, et reposent sur un état de l'art du domaine.

Conclusion. Ce document se termine par une synthèse des apports du travail réalisé, et sur l'ébauche de quelques perspectives intéressantes.

3 Postulats et hypothèses

Les travaux présentés dans cette thèse reposent sur un ensemble de postulats et d'hypothèses.

Définition 1 (Postulat)**.** *Proposition que l'on demande d'admettre comme principe d'une démonstration, bien qu'elle ne soit ni évidente ni démontrée.*

Les postulats que nous posons ne sont donc pas spécialement démontrables. Ils sont cependant le fruit d'une analyse ou d'expérimentations que nous détaillons.

Définition 2 (Hypothèse)**.** *Proposition admise provisoirement avant d'être soumise au contrôle et à la vérification de l'expérimentation.*

Les hypothèses que nous posons nous permettent donc de proposer des solutions, mais sont soumises au verdict de l'évaluation.

Note 1. Nous énonçons ci-dessous les propositions qui sous-tendent la thèse dans sa globalité, tandis que les propositions plus spécifiques, qui concernent un point précis du raisonnement, apparaissent au moment opportun dans le corps du document. Le lecteur qui désirerait consulter rapidement l'ensemble de nos propositions se reportera utilement à l'Annexe E qui les rassemble.

3.1 Diviser pour mieux régner

Notre premier postulat est inspiré de Machiavel, dans son *Art de la guerre* (Mansfield 1993) :

Postulat 1 (Diviser pour mieux régner)**.** *Tout comme un algorithme est une succession d'étapes simples, une tâche complexe n'est bien souvent qu'un ensemble de tâches simples, mais de natures fort différentes. L'identification de ces tâches simples permet dès lors de leur appliquer des solutions simples, inconcevables dans le contexte de la tâche complexe. Il s'agit donc de diviser pour mieux régner.*

Sur la base de ce postulat, nous faisons l'hypothèse suivante :

Hypothèse 1 (Diviser pour mieux régner)**.** *Pour autant qu'une tâche complexe puisse être analysée comme une succession d'étapes simples, les machines à états finis constituent l'outil idéal pour représenter cette succession d'étapes simples de manière aisée.*

3.2 Externalisation des données

Le second postulat s'inscrit dans le contexte d'un système multilingue :

Postulat 2 (Externalisation des données)**.** *En traitement de la langue, une application qui se veut multilingue doit veiller à respecter une séparation stricte entre l'algorithme et les données dépendant de la langue.*

Nous en dégageons l'hypothèse suivante :

Hypothèse 2 (Externalisation des données)**.** *Pour autant que les langues traitées partagent suffisamment de similarités, les machines à états finis autorisent l'externalisation de l'ensemble des traitements dépendant de la langue. Tout l'art réside dès lors dans la détection, au sein d'un processus, des traitements qui ressortissent à la langue.*

Il est important de préciser ce que nous entendons par *multilingue* dans le cadre de cette thèse. Nous n'avons pas la prétention de proposer des algorithmes capables de traiter *toutes* les langues, loin s'en faut.

Les langues du monde se répartissent en familles, selon leur parenté génétique, déterminée à partir de leurs caractéristiques phonétiques, morphologiques, syntaxiques et sémantiques. Les langues d'une même famille partagent donc des caractéristiques linguistiques communes. Elles peuvent cependant se différencier de manière importante sur certains points. *A contrario*, des langues de familles différentes peuvent présenter de nombreuses similarités. C'est ce qui a conduit certains chercheurs à faire l'hypothèse d'une langue originelle dont toutes les langues seraient issues. Le lecteur intéressé se reportera utilement à (Lass 1997, Janda & Joseph 2004).

En tant que francophone, nous désirions évidemment proposer un système capable de traiter au moins le français. En outre, nous voulions que le système puisse accepter toute langue présentant suffisamment de similitudes avec le français. C'est sur cette base que nous avons formulé le Postulat 2 et l'hypothèse qui en découle.

Dans la perspective d'un traitement destiné à un système de synthèse de la parole, les caractéristiques linguistiques du français qui nous paraissent fondamentales sont :

1. Son caractère accentué (cf. Section 8.3.2). Cette caractéristique est déterminante pour le processus de synthèse proposé (cf. Chapitre 10).
2. Son caractère flexionnel (cf. Section 13.1) et sa structure syntaxique, de type SVO (sujet-verbe-objet). Ensemble, ces caractéristiques ont déterminé l'analyse morphologique proposée (cf. Sections 13.4 et 15.5) ainsi que la notion de correction flexionnelle (cf. Section 15.6.2).

Les algorithmes développés dans cette thèse ont donc été pensés pour pouvoir accepter les langues qui partagent ces caractéristiques. Ceci élimine :

1. Les langues tonales, telles que le chinois et le vietnamien, où le ton prime sur l'accent (cf. Section 8.3.2).
2. Les langues agglutinantes, comme le turc, le finnois ou le japonais, qui n'utilisent pas la flexion. Ces langues juxtaposent simplement les éléments de base, sans modification majeure. Contrairement aux langues flexionnelles, les langues agglutinantes expriment généralement les négations, les pronoms et les prépositions sous la forme d'affixes, ce qui n'autorise pas l'emploi de méthodes d'analyse automatique similaires à celles employées dans les langues flexionnelles (György & Csaba 1994).
3. Les langues dont la structure syntaxique diffère de SVO. Par exemple, le turc et le japonais sont de structure SOV (sujet-objet-verbe), l'arabe est de type VSO (verbe-sujet-objet). D'autres structures existent, mais sont minoritaires. Le lecteur intéressé consultera (Comrie 1996).

Les langues ciblées par le système appartiennent, comme le français, au groupe des langues indo-européennes (Beekes 1995). Ce sont :

1. Les langues romanes, issues du latin vulgaire. Les langues romanes se répartissent entre le groupe occidental (français, italien, espagnol, portugais, catalan, etc.), le groupe méridional (corse, sarde) et le groupe oriental (le roumain et ses variantes). Pour une information détaillée à ce sujet, nous renvoyons le lecteur à (Harris & Vincent 1990, Klinkenberg 1994).
2. Les langues germaniques, issues du proto-germanique, une langue reconstituée qui aurait été parlée dans le nord de l'Europe, à l'est du Rhin et du Danube, vers le second âge du fer (500 ACN). Ce groupe fort vaste comprend entre autres des langues occidentales (anglais, allemand et néerlandais) et les langues nordiques (norvégien, suédois, danois et islandais). Pour de plus amples informations, nous renvoyons le lecteur à (Voyles 1992).

Notons que d'autres langues indo-européennes, les langues slaves et les langues grecques, pourraient probablement intégrer les langues cibles du système, parce qu'elles respectent globalement les caractéristiques nécessaires que nous avons mentionnées. Nos connaissances de ces langues sont cependant fort limitées, et l'objectif de cette thèse n'était pas de réaliser une étude linguistique d'un nombre si important de langues. Nous ne les incluons donc pas dans les langues à traiter.

Au sein des langues romanes et germaniques, les différences ne manquent pas :

- Les langues germaniques ont la notion de *particule verbale*.
- L'anglais ne connaît pas de forme polie et ne fait pas varier les adjectifs.
- L'allemand est flexionnel, mais légèrement agglutinant.

- L'allemand et le roumain utilisent encore des déclinaisons, et les langues nordiques ont encore une déclinaison faible de l'adjectif.
- Le roumain, le néerlandais et l'allemand possèdent un genre neutre.
- Le roumain et les langues nordiques utilisent l'article enclitique, tandis que les autres utilisent l'article proclitique.
- etc.

Ces différences ne sont cependant pas de nature à empêcher la mise en place d'un seul et même algorithme : il s'agit de variations morphologiques ou syntaxiques mineures qui, selon notre Hypothèse 2, peuvent être externalisées de l'algorithme grâce aux machines à états finis.

Le système multilingue auquel nous faisons référence est donc un système capable de traiter cet ensemble de langues, sans devoir adapter l'algorithme à la langue. Notons que les évaluations réalisées dans le cadre de cette thèse se sont limitées au français et à l'anglais, langues pour lesquelles nous disposions des données linguistiques nécessaires.

4 Contributions

Les outils et les algorithmes proposés dans le cadre de cette thèse ont été l'occasion d'un certain nombre de contributions aux trois domaines que nous avons abordés : les machines à états finis, la sélection d'unités non uniformes et la correction orthographique.

4.1 Machines à états finis

Nos contributions au domaine s'inscrivent principalement dans le cadre du développement de nos propres outils. Elles se répartissent entre les principes d'implémentation de notre bibliothèque de machines à états finis, et une extension aux algorithmes de compilation des règles de réécriture.

Les travaux que nous présentons contribuent, en outre, à mettre en évidence l'importance fondamentale que revêt la *composition* dans la modélisation d'algorithmes et de modèles en traitement automatique de la langue.

4.1.1 Principes d'implémentation

Notre bibliothèque inclut deux nouveaux modes de représentation des machines à états finis :

1. *Les classes de symboles.* Elles condensent en une seule transition l'ensemble des transitions, de même poids, qui relient deux états d'une machine. Cette originalité est pertinente lorsque le langage modélisé est de la forme $\Sigma^* \alpha$ (cf. Section 6.1.8.1).

2. *Les graphes orientés pondérés.* Les machines peuvent être représentées sous la forme de graphes orientés pondérés, qui font l'économie des symboles qui étiquettent classiquement les transitions entre états. Ce mode de représentation s'applique exclusivement aux langages pensés et conçus comme des graphes, dont les états correspondent aux symboles de l'alphabet (cf. Section 6.1.8.2).

Ces deux modes de représentation ont en commun de réduire la place nécessaire à la représentation des machines, lorsque les langages modélisés respectent les critères mentionnés.

4.1.2 Règles de réécriture

Dans le principe, les règles de réécriture sont relativement faciles à concevoir (cf. Section 5.3). Construire un ensemble de règles exempt d'erreurs de modélisation est cependant un exercice délicat.

Afin de faciliter l'expression de contraintes et d'éviter les erreurs de modélisation, nous avons défini la notion de *marqueur*, qui permet d'identifier un phénomène et d'en suivre l'évolution (cf. Section 6.2.4). Le marqueur peut être :

1. Un *déclencheur*, qui indique de manière non ambiguë qu'une condition d'application a été rencontrée.
2. Un *masqueur*, qui évite l'application erronée d'une règle sur une expression régulière.
3. Un *bloqueur*, qui empêche la formation d'une expression régulière sur laquelle une règle pourrait s'appliquer à tort.

Notre compilateur permet l'utilisation de ces marqueurs, et propose un mécanisme d'inclusion de fichiers (cf. Section 6.2.2.3). Ensemble, ces deux caractéristiques du compilateur facilitent fortement la construction de modèles complexes. Cette contribution s'inscrit donc résolument dans le domaine de la formalisation des langages.

4.1.3 Importance de la composition

La composition est une opération associative applicable aux transducteurs à états finis. Elle permet la conception de relations complexes à partir de relations simples (cf. Section 3.3.3).

Les modèles que nous proposons en sélection d'unités non uniformes (cf. Section 10.2.4), en analyse morpho-syntaxique (cf. Section 15.5) et en correction orthographique (cf. Sections 15.6 et 16.4) sont complexes, mais ont été construits par composition, à partir de modèles beaucoup plus simples à concevoir.

Nous avons exploité l'associativité de la composition afin d'assurer l'adaptabilité, l'efficacité et la flexibilité de nos modèles :

1. *Adaptabilité.* Les différents composants d'un même modèle peuvent être dispersés au sein d'un processus. Ceci facilite l'adaptation d'un composant au type d'une donnée (cf., par exemple, Section 15.5).

2. *Efficacité.* Avant d'être composés ensemble, les composants du modèle peuvent être simplifiés par projection et déterminisation, ce qui réduit l'espace mémoire et le temps de calcul nécessaires (cf., par exemple, Section 16.4.3).
3. *Flexibilité.* Un composant d'un modèle peut être recalculé indépendamment des autres composants. Ceci facilite l'estimation de l'importance et de l'impact d'un composant sur le modèle global.

La composition contribue de ce fait à la mise en œuvre de notre Postulat 1, qui pose l'intérêt de diviser pour mieux régner.

4.2 Sélection d'unités non uniformes

La sélection des unités de parole se divise traditionnellement en deux phases distinctes (cf.Section 8.5.2) :

1. L'estimation d'un coût cible, qui détermine la distance entre une unité cible et les unités candidates du corpus de parole.
2. L'estimation d'un coût de concaténation, qui évalue la distance entre deux unités voisines dans le flux de parole.

Dans ce contexte, nos contributions concernent la répartition des critères entre le coût cible et le coût de concaténation, le choix des critères linguistiques et la définition d'une méthode de pondération automatique du coût cible.

4.2.1 Répartition des critères de sélection

Nous démontrons l'intérêt de déterminer le coût cible à l'aide de critères exclusivement linguistiques, et d'évaluer le coût de concaténation sur la base de critères exclusivement acoustiques (cf.Sections 10.1.1.1, 10.1.7 et 10.2.7).

4.2.2 Choix des critères linguistiques

Les critères linguistiques suivants sont indéniablement pertinents dans l'établissement d'un coût cible de qualité (cf.Sections 10.1.1.3, 10.1.3 et 10.2.2) :

1. Le diphone, comme unité de sélection.
2. L'accent de mot, que nous conservons en surplus de l'accent de groupe.
3. La position dans le groupe rythmique.
4. La distance par rapport à la pause.

4.2.3 Méthode de pondération automatique du coût cible

Considérant que les critères linguistiques du coût cible participent à une prise de décision prosodique, nous avons proposé de les pondérer par *entropie*. Plus précisément,

nous déterminons le *rapport de gain* de chaque critère linguistique. Dans l'établissement du coût cible, le critère prépondérant est celui qui présente le meilleur rapport de gain. Les critères retenus participent donc à l'élaboration du coût cible dans l'ordre décroissant de leur rapport de gain (cf.Section 10.1.5).

4.3 Correction orthographique

La correction orthographique que nous proposons en synthèse de la parole est réalisée au cours de l'analyse morpho-syntaxique. Dans ce contexte, nos contributions principales se répartissent entre l'analyse morpho-syntaxique elle-même et la modélisation des erreurs.

4.3.1 Analyse morpho-syntaxique

Dans le module de traitement automatique de la langue dans lequel nous avons intégré nos algorithmes, l'analyse syntaxique était réalisée par un modèle de langue. Classiquement, le modèle de langue est réduit à un modèle syntaxique, et perd de ce fait toute référence au mot.

Afin de réintroduire le mot dans l'estimation statistique de la phrase, nous proposons un modèle d'ambiguïté lexicale. Dans un premier temps, ce modèle a simplement permis de remplacer le mot par sa classe d'ambiguïté lexicale, lorsque le mot n'avait pas été rencontré dans le corpus d'apprentissage (cf.Section 13.5.3). Dans un deuxième temps, le modèle a été étendu, afin de permettre une estimation adaptée selon que le mot appartient ou non au vocabulaire connu (cf.Section 15.5).

4.3.2 Modélisation des erreurs

Nos contributions concernent la correction des mots hors-vocabulaire :

1. *Flexibilité.* La méthode de correction que nous proposons est une distance d'édition flexible : elle tient compte des caractéristiques de la forme à corriger pour déterminer non seulement le type d'erreurs acceptées, mais également le nombre d'erreurs autorisées (cf.Section 15.6.1). Les distances d'édition de l'état de l'art, par contre, sont contraintes de fixer le nombre d'erreurs *a priori*, quelle que soit la longueur de la forme (cf.Section 14.4.1).
2. *Gestion des erreurs $n \rightarrow m$.* En reconnaissance optique des caractères, les erreurs de segmentation de l'image sont susceptibles de produire des erreurs d'alignement entre les caractères reconnus et les caractères attendus : nous sommes dans le cas d'erreurs de type $n \rightarrow m$. Dans ce contexte, nous montrons que des règles de réécriture permettent très simplement de modéliser des erreurs $n \rightarrow m$, que celles-ci aient été décrites par des experts ou apprises à partir d'un entraînement. Ce modèle est le seul qui gère les erreurs $n \rightarrow m$ tout en restant indépendant du contexte (cf.Section 16.4).

Première partie

Machines à états finis

Chapitre 1

Introduction

En l'absence d'une bibliothèque de machines à états finis qui réponde aux besoins des applications que nous présentons dans les Parties II et III de ce document, nous avons développé notre propre bibliothèque et notre propre compilateur d'expressions régulières et de règles de réécriture pondérées. Ces outils implémentent les algorithmes de l'état de l'art, et proposent quelques contributions personnelles, qui concernent les modes de représentation des machines à états finis et la formalisation des règles de réécriture.

L'objectif de cette partie est de rassembler en un seul lieu ce qui fonde l'ensemble de nos applications et en constitue le point de départ : la théorie des langages formels, les concepts et les algorithmes relatifs aux machines à états finis, les principes des expressions régulières et des règles de réécriture, et les outils que nous avons développés.

1.1 Plan de la partie

Chapitre 1 : rappel de quelques notions fondamentales, et situation des FSMs dans le domaine plus général des langages formels.

Chapitres 2 à 4 : présentation des automates (Chapitre 2), des transducteurs (Chapitre 3) et de leurs correspondants pondérés (Chapitre 4). Nous décrivons les propriétés de ces machines, et les principales opérations qui s'y rapportent. Pour chaque opération, nous détaillons le principe de l'algorithme, que nous illustrons par un pseudocode qui se veut un reflet fidèle de l'implémentation dans notre bibliothèque. L'idée est de permettre au lecteur de distinguer, dans le pseudocode, les structures efficaces à mettre en œuvre.

Chapitre 5 : présentation des expressions régulières et des règles de réécriture. Dans ce chapitre, l'accent est mis sur les algorithmes de compilation des expressions régulières et des règles de réécriture en machines à états finis.

Chapitre 6 : présentation des principes informatiques qui ont régi le développement de nos outils, et des extensions que le développement de ces outils nous a permis de proposer.

Chapitre 7 : mise en évidence des points importants abordés dans cette partie.

1.2 Notions fondamentales

Alphabet. Un alphabet est un ensemble généralement fini de symboles que nous notons Σ. En voici quelques exemples :

- $\Sigma_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ est l'alphabet des 10 chiffres arabes.
- $\Sigma_2 = \{a, b, \ldots, y, z, A, B, \ldots, Y, Z\}$ est l'alphabet latin, minuscules et majuscules.
- $\Sigma_3 = \{\text{Adj, Adv, Conjcoor, Det, Noun, Verb, Prep}\}$ est un alphabet (fort réduit) de catégories syntaxiques.

Concaténation et strings. A partir des symboles d'un alphabet Σ donné, la *concaténation* permet de construire des *séquences de symboles*. Notons que les séquences de symboles sont appelées *chaînes* dans la littérature francophone et *strings* dans la littérature anglophone. Dans ce document, nous employons le terme *string*, parce qu'il nous paraît moins ambigu que *chaîne*[1] et qu'il fait également référence à la notion de *séquence de caractères* en programmation.

Si a et b sont des symboles d'un alphabet Σ, la concaténation de a et b se note classiquement $a \cdot b$. De même, si x et y sont des strings construites sur Σ, la concaténation de x et y se note $x \cdot y$.

Formellement, si $x = a_1a_2a_3 \cdots a_i$ et $y = b_1b_2b_3 \cdots b_j$, alors $x \cdot y$ est la string de longueur $i + j$: $xy = a_1a_2a_3 \cdots a_ib_1b_2b_3 \cdots b_j$.

Remarquons que l'opérateur de concaténation est souvent omis : $a{\cdot}b = ab$, $x{\cdot}y = xy$.

String vide. La string qui ne contient aucun caractère est appelée *string vide* ou *epsilon*, et se note ϵ. La string vide est l'élément neutre pour la concaténation : $x \cdot \epsilon = \epsilon \cdot x = x$.

Monoïde libre. L'ensemble des strings construites à partir de Σ se note Σ^* et est appelé le *monoïde libre*.

Un *monoïde* est une structure algébrique $(E, +, 0)$ consistant en un ensemble E muni d'une loi de composition interne associative $+$ et d'un élément neutre 0. Dans le cas du monoïde libre, la loi de composition est la concaténation, et l'élément neutre est ϵ : $(\Sigma, \cdot, \epsilon)$.

Le monoïde libre Σ^*, construit sur l'ensemble Σ, est donc le monoïde dont les éléments sont toutes les strings de zéro, un ou plusieurs symboles de Σ. Par exemple, si $\Sigma = \{a, b\}$, $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aab, \cdots\}$.

Langage. Un langage L est un ensemble de strings définies sur un alphabet Σ donné. Par définition, L est donc un sous-ensemble de Σ^* : $L \subseteq \Sigma^*$.

[1] Ce terme pourrait par exemple faire involontairement référence aux listes chaînées.

1.3 Langages formels et hiérarchie de Chomsky

1.3.1 Langages formels, grammaires formelles

On parle de *langage formel* lorsqu'un langage peut être décrit à l'aide d'une *grammaire formelle*.

Définition 1.3.1. *Une* grammaire formelle *G est un 4-uplet (S, N, T, R) où S est le symbole initial appelé* axiome*, N est un alphabet fini de symboles non-terminaux, T est un alphabet fini de symboles terminaux et R est un ensemble de règles de production dont les membres gauche et droit sont des strings formées de non-terminaux et de terminaux.*

Voici un exemple de règle de production :

$$S \rightarrow aSb$$

Appliquer une règle de production consiste à *réécrire* son membre de gauche à l'aide de son membre de droite. La flèche $\rightarrow$ peut d'ailleurs se lire « se réécrit ». L'application successive d'un certain nombre de règles de production s'appelle une *dérivation*.

Dans une grammaire formelle, les symboles terminaux sont les symboles du langage décrit.

Définition 1.3.2. *Le* langage formel *défini par une grammaire formelle $G = (S, N, T, R)$ est l'ensemble des strings formées uniquement de symboles terminaux qui peuvent être atteints par dérivation à partir de l'axiome de la grammaire.*

La Table 1.1 donne un exemple classique de règles de production, sur lequel nous reviendrons dans la suite de ce document. Ces règles sont définies sur $N = \{A, B\}$ et $T = \{$a, b$\}$. Nous notons les non-terminaux en majuscules, les terminaux en minuscules :

(1)	S	$\rightarrow$	$A\ S\ B$
(2)	S	$\rightarrow$	ϵ
(3)	A	$\rightarrow$	a
(4)	B	$\rightarrow$	b

TAB. 1.1: Exemple de grammaire formelle

Parmi les règles de la Table 1.1, la règle (2) permet de mettre fin à la récursivité autorisée par la règle (1). Ces règles décrivent le langage $a^n b^n$, c'est-à-dire le langage dans lequel un nombre donné de a est systématiquement suivi du même nombre de b.

Il est intéressant de noter qu'une grammaire formelle peut être utilisée en *génération* ou en *analyse*.

En mode *analyse*, l'application des règles de dérivation à une string donnée peut arriver à une situation de blocage, où des éléments non-terminaux n'ont pas encore été réécrits, mais où plus aucune règle ne peut s'appliquer à la string en l'état. On considère dans ce cas que la string n'appartient pas au langage décrit par la grammaire. La string *aaabb* par exemple, analysée par les règles de la Table 1.1, arrivera à cette situation de blocage, après l'application des règles dans l'ordre suivant : (1) ASB, (3) aSB, (1) a$ASBB$, (3) aaSBB, (1) aa$ASBBB$, (3) aaa$SBBB$, (2) aaaϵBBB, (4) aaaϵbBB, (4) aaaϵbb$B$.

1.3.2 Hiérarchie de Chomsky

Dans le cadre de son étude des structures syntaxiques du langage naturel et en prélude à la définition de la théorie de la *grammaire générative*, Chomsky a étudié les langages formels et leur capacité à modéliser le langage humain. Dans son célèbre article (Chomsky 1956), il propose une classification des grammaires formelles connue aujourd'hui sous le nom de *hiérarchie de Chomsky*.

La hiérarchie de Chomsky répartit les grammaires formelles en cinq catégories, du type 0 au type 4. Chaque type i a un pouvoir d'expression plus large que le type $i+1$, de sorte que les langages de type i *incluent* logiquement les langages de types $i+1$.

Notons que, dans la description de sa hiérarchie, Chomsky détermine le type de machine informatique qui permet de reconnaître un type de grammaire donné. L'objectif de cette section n'est pas de revoir par le détail tous les types de machines existants, mais de situer les FSMs dans la hiérarchie des langages. Les autres types de machines ne sont donc pas décrits dans ces pages, et nous invitons le lecteur intéressé à consulter (Aho *et al.* 1974, Hopcroft *et al.* 1979).

Type 0 : grammaires non restreintes

$$w_1 \rightarrow w_2 \quad \text{avec} \quad w_1, w_2 \in (N \cup T)^*$$

En d'autres termes :

$$(N \cup T)^* \rightarrow (N \cup T)^*$$

Par définition, ces grammaires ne sont limitées par aucune contrainte. Les langages qu'elles reconnaissent sont qualifiés de *récursivement énumérables*. Pour rappel, un ensemble est dit récursivement énumérable s'il existe un procédé mécanique qui peut dire si un élément appartient à l'ensemble. Le même procédé, cependant, ne peut rien dire si l'élément n'appartient pas à l'ensemble.

Ces langages, très puissants, sont exactement ceux reconnus par une machine de Turing, qui boucle à l'infini lorsqu'une string n'appartient pas au langage accepté. Ces langages sont dès lors trop puissants pour être exploités.

Type 1 : grammaires contextuelles

$$l\,X\,r \rightarrow l\,w\,r \quad \text{avec} \quad \begin{array}{l} X \in N \\ w \in (N \cup T)^+ \\ l, r \in (N \cup T)^* \end{array}$$

En d'autres termes :

$$(N \cup T)^*\, N\, (N \cup T)^* \rightarrow (N \cup T)^*\, (N \cup T)^+\, (N \cup T)^*$$

On parle de grammaires contextuelles parce que la réécriture d'un symbole non-terminal (X) *peut* dépendre de son contexte (l et r). La notion de contexte ainsi que l'obligation pour la réécriture (w) d'être non vide assure que la partie droite de la règle de production est *au moins* aussi longue que la partie gauche.

Les langages produits par ces grammaires sont les langages contextuels, et sont exactement ceux reconnus par machines de Turing linéairement bornées non déterministes. Ces machines ne bouclent pas à l'infini lorsqu'une string n'appartient pas au langage accepté.

Type 2 : grammaires hors-contexte

$$X \rightarrow w \quad \text{avec} \quad \begin{array}{l} X \in N \\ w \in (N \cup T)^* \end{array}$$

En d'autres termes :

$$N \rightarrow (N \cup T)^*$$

Ce type de grammaires traite les symboles non-terminaux séparément, et donc hors-contexte. Les grammaires hors-contexte autorisent les imbrications infinies, étant donné que la réécriture d'un symbole non-terminal X peut contenir X lui-même. Les règles de la Table 1.1 constituent un exemple de grammaire hors-contexte.

Les langages produits par ces grammaires sont les langages non-contextuels, et sont exactement ceux reconnus par les automates à pile.

Type 3 : grammaires rationnelles

L'ensemble des grammaires rationnelles est composé des grammaires linéaires à gauche et des grammaires linéaires à droite.

- Grammaires linéaires à gauche :

$$\begin{array}{lll} X \to a & \text{avec} & X \in N \\ & & a \in T^+ \\ X \to Ya & \text{avec} & X, Y \in N \\ & & a \in T^* \end{array}$$

  En d'autres termes :

$$\begin{array}{l} N \to T^+ \\ N \to T^* N \end{array}$$

- Grammaires linéaires à droite :

$$\begin{array}{lll} X \to a & \text{avec} & X \in N \\ & & a \in T^+ \\ X \to aY & \text{avec} & X, Y \in N \\ & & a \in T^* \end{array}$$

  En d'autres termes :

$$\begin{array}{l} N \to T^+ \\ N \to N\,T^* \end{array}$$

Une grammaire rationnelle est donc linéaire à gauche *ou* linéaire à droite. Du fait de cette restriction, ces grammaires n'autorisent pas les imbrications infinies, contrairement aux grammaires hors-contexte.

Les langages produits par ces grammaires sont les langages rationnels ou réguliers, et sont exactement ceux reconnus par les automates à états finis. C'est donc à ce niveau de la hiérarchie de Chomsky que se situent les FSMs dans leur ensemble. L'équivalence des automates à états finis et des langages réguliers est détaillée dans le Chapitre 2.

Type 4 : grammaires à choix finis

$$\begin{array}{lll} X \to a & \text{avec} & X \in N \\ & & a \in T^+ \end{array}$$

En d'autres termes :

$$N \to T^+$$

Cette classe est peu productive, étant donné qu'un non-terminal se réécrit exclusivement par un ou plusieurs terminaux. Les langages reconnus par ces grammaires sont les langages finis.

1.3.3 Langages naturels, langages non-contextuels ?

Au contraire de Shannon (1948) qui a démontré la possibilité d'une *approximation du langage naturel* à l'aide de modèles de type rationnel, l'objectif de Chomsky, en posant sa hiérarchie des langages, est de démontrer l'incapacité des langages rationnels à gérer le niveau syntaxique des langages naturels.

L'argument sur lequel Chomsky fonde son rejet des langages rationnels est la présence, dans les langages naturels, de structures parenthésées provenant de constructions grammaticales comme :

« si S_1 alors S_2. »
« soit S_1, ou S_2. »
« L'homme qui a dit que S_n arrive aujourd'hui. »

qui impliquent des dépendances : *si* et *alors*, *soit* et *ou*, *homme* et *arrive*. Ce type de structures, que l'on trouve également dans les langages de programmation, autorise les imbrications infinies.

A propos de l'anglais qui était son sujet d'étude, Chomsky conclut que « *English is not a finite-state language* » (Chomsky 1956). Plus largement, dans (Chomsky & Schützenberger 1963), il formalise les grammaires formelles et conclut que la syntaxe des langues nécessite de recourir aux grammaires hors-contexte, tandis que leur morphologie, qui n'implique pas d'imbrications infinies, peut être modélisée à l'aide de langages réguliers.

Cette distinction posée par Chomsky a été largement appliquée au niveau des langages de programmation, dont le niveau lexical est géré par des automates à états finis, tandis que le niveau syntaxique est géré par des automates à pile. Cette distinction est également à l'origine d'un rejet massif, de la part des linguistes informaticiens, des langages rationnels au profit des langages non-contextuels pour la modélisation du langage naturel.

Pourtant, dès la rédaction de son article fondateur, Chomsky reconnaît que les imbrications infinies ne sont que théoriques dans le langage naturel : l'être humain n'est pas capable d'appréhender le sens de structures fortement imbriquées, comme c'est le cas par exemple dans la phrase de la Figure 1.1 :

S_1	L'homme			part.
S_2		que la femme	aime	
S_3			que le chien regarde	

FIG. 1.1: Structures fortement imbriquées

Chomsky malgré tout conclut ceci : « *If the processes have a limit, then the construction of a finite-state grammar will not be literally impossible (since a list is a trivial finite-state grammar), but this grammar will be so complex as to be of little interest.* »

Chomsky, en rejetant de la sorte les langages réguliers au profit des langages non-contextuels, néglige les avantages que seuls les langages réguliers proposent :

- De nombreuses opérations mathématiques, dont les opérations définies sur les ensembles, sont applicables aux langages réguliers seuls.
- La théorie logique associée aux langages réguliers est décidable, alors que la plupart des problèmes associés aux grammaires ne le sont pas.
- Les FSMs ne se limitent pas aux automates, mais comportent également les transducteurs, qui définissent des relations entre langages rationnels et autorisent, grâce à l'opération de composition, la modélisation de langages complexes à partir de langages simples.

Ce sont ces avantages qui ont poussé Gross & Perrin (1989) à considérer que les FSMs fournissent l'essentiel de la description des langages naturels.

Les travaux de Schützenberger (1961, 1977) et de Kuich & Salomaa (1986) ont en outre démontré que les FSMs permettent de représenter et de manipuler des langages pondérés, et donc de gérer une certaine *incertitude*.

Mohri (2002b) a d'ailleurs récemment proposé des extensions aux algorithmes définis sur les automates et les transducteurs applicables aux automates et transducteurs pondérés. Ces extensions ont ouvert la voie à une nouvelle technologie du langage basée sur des FSMs pondérés, là où auparavant *modèles de langue* rimaient inéluctablement avec *programmation dynamique* ou *modèles de Markov cachés*.

Chapitre 2

Les automates

2.1 Introduction

Les automates, considérés comme une classe de graphes ou de langages, sont des outils puissants. D'une part, ils autorisent différents niveaux de représentation qui facilitent leur construction et optimisent leur manipulation. D'autre part, ils acceptent une opération de minimisation qui permet de réduire la taille requise pour les représenter. Enfin et peut-être surtout, ils sont équivalents aux langages réguliers et acceptent dès lors les opérations définies sur ces langages.

Après avoir rappelé les principales définitions relatives aux automates, cette section détaille leurs niveaux de représentation et les algorithmes qui permettent de passer d'un niveau de représentation à un autre. La section se poursuit avec la description de l'algorithme de minimisation, basé sur la notion d'équivalence, et présente enfin les opérations régulières applicables aux automates.

2.2 Définitions

Les automates peuvent être considérés comme définissant une classe de graphes ou une classe de langages.

2.2.1 Graphes

Selon la première interprétation, les automates peuvent être vus comme des graphes orientés dont les arcs sont étiquetés.

Définition 2.2.1. *Un automate à états finis A est un 5-uplet (Σ, Q, i, F, E) où Σ est un ensemble fini de symboles appelé alphabet, Q est un ensemble fini d'états, $i \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux, et $E \subseteq Q \times \Sigma \times Q$ est un ensemble fini de transitions de la forme (p, a, q).*

Remarquons que l'ensemble E peut être remplacé par une fonction de transition δ qui projette $Q \times \Sigma$ sur Q. Il y a équivalence des deux définitions :

$$\delta(p, a) = q \in Q \mid \exists (p, a, q) \in E \tag{2.2.1.1}$$

Convention 2.2.1. Dans ce document, les graphes sont représentés selon les conventions suivantes :

- l'état initial du graphe est toujours 0,
- tout état final se représente par un double cercle, les autres états par un simple cercle.

Par exemple, concernant l'automate présenté en Figure 2.1, $\Sigma = \{\text{a,b}\}$, $Q = \{0, 1\}$, $i = 0$, $F = \{1\}$ et $E = \{(0, \text{a}, 0), (0, \text{b}, 1), (1, \text{b}, 1)\}$.

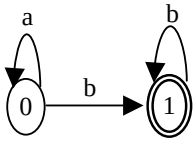


FIG. 2.1: Automate à états finis A

Définition 2.2.2. *Un* chemin *dans un graphe est une séquence de transitions de la forme* (p_0, a_1, p_1), (p_1, a_2, p_2), $\ldots$, (p_{n-1}, a_n, p_n). *Un chemin est dit* simple *si toutes les transitions et tous les états du chemin sont distincts, excepté le premier et le dernier états qui peuvent être identiques.*

Remarquons que le graphe de la Figure 2.1 présente des transitions entrant dans leur état d'origine. Ceci amène à définir la notion de *cycle*.

Définition 2.2.3. *Un* cycle *est un chemin simple de longueur au moins égale à 1 qui commence et termine au même état. Un graphe est dit cyclique s'il contient au moins un cycle. Dans le cas contraire, un graphe est dit acyclique.*

2.2.2 Langages

Selon la deuxième interprétation, les automates définissent une classe de langages. Cette interprétation demande préalablement d'étendre l'ensemble de transitions E de sorte qu'il opère sur des strings : l'ensemble étendu de transitions $\hat{E} \subseteq Q \times \Sigma^* \times Q$ est défini comme le plus petit ensemble tel que :

(i) $\forall q \in Q, (q, \epsilon, q) \in \hat{E}$

(ii) $\forall u \in \Sigma^*$ et $\forall a \in \Sigma$, si $(p, u, q) \in \hat{E}$ et $(q, a, r) \in E$,
alors $(p, u \cdot a, r) \in \hat{E}$

La fonction δ peut être étendue de manière similaire. La fonction de transition étendue $\hat{\delta}$ est la fonction qui projette $Q \times \Sigma^*$ sur Q de sorte que :

(i) $\forall q \in Q$, $\hat{\delta}(q, \epsilon) = \{q\}$

(ii) $\forall u \in \Sigma^*$ et $\forall a \in \Sigma$, $\hat{\delta}(p, u \cdot a) = \delta(\hat{\delta}(p, u), a)$

Ces extensions étant faites, un automate A peut maintenant être considéré comme définissant un langage $L(A)$.

Définition 2.2.4. *Le* langage d'un automate $A = (\Sigma, Q, i, F, \hat{\delta})$ *est l'ensemble des* strings*, construites sur Σ, pour lesquelles un chemin existe de l'état initial à un état final :*

$$L(A) = \{u \in \Sigma^* : \hat{\delta}(i, u) \cap F \neq \emptyset\} \tag{2.2.2.1}$$

2.3 Niveaux de représentation des automates

Selon sa fonction de transition et les symboles qu'il accepte, un automate est *déterministe*, *non-déterministe* ou *non-déterministe avec transitions ϵ*.

Ces différents niveaux de représentation, qui sont dits *équivalents* parce qu'ils permettent de définir le même langage, ont chacun leur intérêt dans la construction et la manipulation des automates.

2.3.1 Automate déterministe

L'automate décrit dans la Définition 2.2.1 est un automate déterministe (DFA). Dans un DFA (cf. Figure 2.2), aucune transition n'est étiquetée par le symbole ϵ et la fonction de transition δ pour une paire {état p, symbole a} donnée détermine un et un seul état q. A elle seule, cette propriété rend les automates bien plus efficaces que de nombreux autres

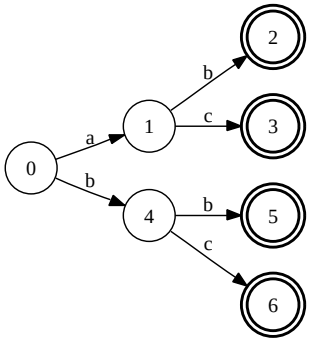


FIG. 2.2: Automate déterministe A_D

outils informatiques : elle certifie que le temps de parcours d'une string est linéairement proportionnel à la longueur de la string et ne dépend en aucun cas de la taille du langage de l'automate. Pour une string u de longueur $|u|$ et un automate A de taille $|A|$, le temps

de parcours est donc de complexité $O(|u|)$.

Un automate déterministe peut en outre être *complet*. La Figure 2.3 en donne un exemple.

Définition 2.3.1. *Un état d'un automate déterministe est qualifié de* complet *s'il possède exactement une transition pour tout symbole de l'alphabet sur lequel l'automate est défini. Un automate déterministe est complet si tous ses états sont complets.*

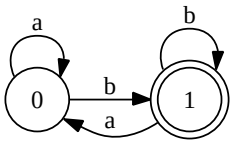


FIG. 2.3: Automate déterministe complet défini sur $\Sigma = \{a, b\}$

2.3.2 Automate non-déterministe

Définition 2.3.2. *Un automate non-déterministe (NFA) A est un 5-uplet $(\Sigma, Q, i, F, \delta)$ où Σ est un ensemble fini de symboles appelé alphabet, Q est un ensemble fini d'états, $i \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux, et δ est la fonction de transition qui projette $Q \times \Sigma$ sur $\mathcal{P}^Q$.*

Dans un NFA donc, l'image de la fonction de transition δ pour une paire {état p, symbole a} donnée peut être un ensemble d'états, mais les transitions entre états ne peuvent être étiquetées par le symbole ϵ. La Figure 2.4 donne un exemple d'automate non-déterministe.

En terme de complexité, le parcours d'une string u de longueur $|u|$ dans un automate A de taille $|A|$ est quadratique si A est non déterministe ($O(|u| \times |A|)$), alors qu'il est linéaire si A est déterministe ($O(|u|)$). Informellement, un NFA n'assure donc pas que le temps de parcours d'une string soit linéairement proportionnel à la longueur de la string. Cependant, pour un NFA $A_N = (\Sigma, Q, i, F, \delta)$ donné, il est toujours possible de calculer un DFA équivalent $A_D = (\Sigma, Q', i', F', d')$.

L'algorithme qui calcule le DFA équivalent à un NFA donné est la *déterminisation*, présentée en Pseudocode 1. Cet algorithme se base sur l'idée que chaque état de la machine déterministe correspond à un ensemble d'états de la machine non déterministe. De ce fait, l'algorithme construit de proche en proche l'ensemble des parties de Q, à partir de l'état initial.

Plus formellement, la déterminisation appliquée à $A_N = (\Sigma, Q, i, F, \delta)$ construit $A_D = (\Sigma, Q', i', F', d')$ tel que :

- $i' = \{i\}$ (ligne 2)
- $Q' \subseteq \mathcal{P}^Q$ (lignes 3 et 18)

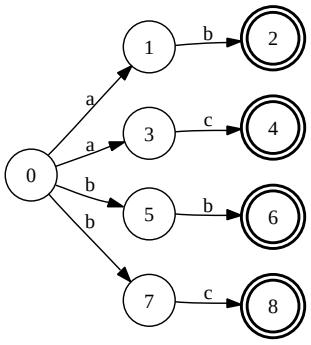


FIG. 2.4: Automate non-déterministe A_N

Require: Un NFA $A_N = (\Sigma, Q, i, F, E)$
Ensure: Le DFA $A_D = (\Sigma, Q', i', F', E')$ équivalent à A_N

$H \leftarrow F' \leftarrow E' \leftarrow \emptyset$
$p' \leftarrow i' \leftarrow$ `INSERT`$(H, \{i\})$ /* 1^{er} élément $\rightarrow$ retour $= 0$ */
$Q' \leftarrow \{i'\}$
while $p' < |Q'|$ **do**
  $S_{p'} \leftarrow$ `GET`(H, p')
  if $F \cap S_{p'} \neq \emptyset$ **then**
    $F' \leftarrow F' \cup \{p'\}$
  end if
  for each $a \in \Sigma$ **do**
    $S_{q'} \leftarrow \emptyset$
    for each $p \in S_{p'}$ **do**
      for each $e \in E[p] : e.a = a$ **do**
        $S_{q'} \leftarrow S_{q'} \cup \{e.q\}$
      end for
    end for
    if $S_{q'} \neq \emptyset$ **then**
      $q' \leftarrow$ `INSERT`$(H, S_{q'})$
      $Q' \leftarrow Q' \cup \{q'\}$
      $E' \leftarrow E' \cup \{(p', a, q')\}$
    end if
  end for
  $p' = p' + 1$
end while
$A_D \leftarrow (\Sigma, Q', i', F', E')$
return A_D

Pseudocode 1: `DETERMINISATION`

- $F' = \{q' \in Q' : q' \cap F \neq \emptyset\}$ (lignes 6–8)
- $d'(q, a) = \bigcup_{q \in q'} \delta(q, a)$ (lignes 12–14 et 19)

La déterminisation de l'automate présenté en Figure 2.4 donnera l'automate de la Figure 2.5, équivalent au DFA présenté en Figure 2.2.

2.3.3 Automate non-déterministe avec transitions ϵ

Définition 2.3.3. *Un automate non-déterministe avec transitions ϵ (ϵ-NFA) A est un 5-uplet $(\Sigma, Q, i, F, \delta)$ où Σ est un ensemble fini de symboles appelé alphabet, Q est un ensemble fini d'états, $i \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux, et δ est la fonction de transition qui projette $Q \times (\Sigma \cup \{\epsilon\})$ sur $\mathcal{P}^Q$.*

La Figure 2.6 donne un exemple d'automate non-déterministe avec transitions ϵ. Les ϵ-NFAs ont un intérêt incontestable. En effet, comme nous le montrons dans le Chapitre 3 et comme nous le détaillons dans la Section 5.2, un ϵ-NFA peut être le résultat d'opérations réalisées sur un ou plusieurs FSMs. Cependant, dans un ϵ-NFA,

- une string d'entrée peut donner lieu à plusieurs chemins dans le graphe, étant donné que l'image de la fonction de transition δ pour une paire {état p, symbole a} donnée peut être un ensemble d'états : $Q \times (\Sigma \cup \{\epsilon\}) \longrightarrow \mathcal{P}^Q$.
- une transition entre deux états p et q peut être étiquetée avec la string vide ϵ, étant donné que l'ensemble de symboles sur lequel sont définis E et δ est $(\Sigma \cup \{\epsilon\})$.

Quelle que soit la raison de la présence de transitions ϵ, ces transitions induisent donc un *retard* lors du parcours d'une string d'entrée dans l'automate. Pour une utilisation efficace de l'automate, il est donc nécessaire de les supprimer.

Plusieurs algorithmes permettent de construire un automate sans transitions ϵ équivalent à un automate donné présentant des transitions ϵ. On parle d'un algorithme *suppression-ϵ*.

Quel que soit l'algorithme de suppression-ϵ, celui-ci repose sur le concept de *clôture-ϵ* d'un état. La clôture-ϵ d'un état p contient p lui-même ainsi que l'ensemble des états accessibles en suivant tous les chemins étiquetés ϵ à partir de p. La clôture-ϵ de p se construit donc de manière récursive : si un état q donné est accessible par une transition ϵ à partir de p, la clôture-ϵ de p contiendra la clôture-ϵ de q, définie récursivement comme celle de p. Le Pseudocode 2 présente l'algorithme de clôture-ϵ.

L'algorithme de suppression-ϵ est classiquement présenté en combinaison avec la déterminisation, de sorte que l'ϵ-NFA est directement converti en DFA (Aho *et al.* 1986). Cependant, Mohri *et al.* (2001) ont montré que dans le cas de machines pondérées, la déterminisation n'est pas toujours calculable (cf. Section 4.6.2). Pour cette raison, le Pseudocode 3

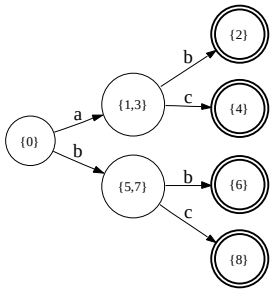


FIG. 2.5: Automate obtenu par déterminisation de A_N

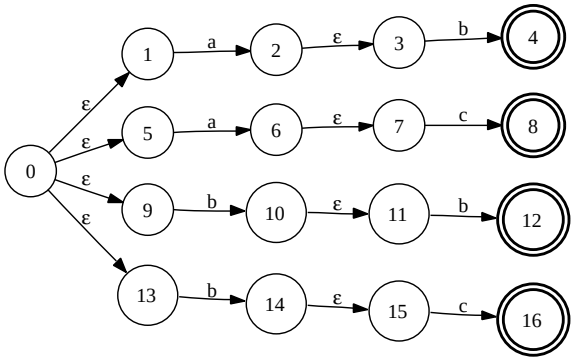


FIG. 2.6: Automate non-déterministe avec transitions ϵ A_ϵ

Require: Q_ϵ, un vecteur d'ensembles de taille $|Q|$;
p, un état de Q
Ensure: $Q_\epsilon[p]$ contient la clôture-ϵ de p
1: **if** $Q_\epsilon[p] = \emptyset$ **then**
2: $Q_\epsilon[p] \leftarrow \{p\}$
3: **for each** $e \in E[p] : e.a = \epsilon$ **do**
4: `CLOTURE-EPS`$(Q_\epsilon, e.q)$
5: $Q_\epsilon[p] \leftarrow Q_\epsilon[p] \cup Q_\epsilon[e.q]$
6: **end for**
7: **end if**

Pseudocode 2: `CLOTURE-EPS`

présente une suppression-ϵ qui, pour un ϵ-NFA $A_\epsilon = (\Sigma, Q, i, F, E)$, construit uniquement le NFA équivalent $A_N = (\Sigma, Q', i', F', E')$. L'algorithme commence en calculant la clôture-ϵ de tous les états de A_ϵ (ligne 2). Ensuite, i' est initialisé comme la clôture-ϵ de i (ligne 4), et ajouté dans Q' (ligne 5). L'algorithme construit ensuite les états de Q' de proche en proche, à partir de i' (lignes 6–19). Pour un ensemble S correspondant à un état de Q', S est final s'il contient au moins un état $p \in F$ (lignes 8–10). Ensuite, pour chaque transition étiquetée par un symbole de l'alphabet et quittant un état $p \in S$, la clôture-ϵ de l'état atteint par la transition est ajouté comme état de Q' (lignes 13–14) et l'ensemble des transitions E' est mis à jour (ligne 15).

Appliqué à l'automate présenté en Figure 2.6, cet algorithme donnera l'automate de la Figure 2.7, équivalent au NFA présenté en Figure 2.4.

2.4 Equivalence et minimisation

Nous venons de montrer que plusieurs niveaux de représentation, du ϵ-NFA au DFA, permettent de définir le même langage et sont dans ce cas dits *équivalents*. Nous avons également montré que des algorithmes permettent de passer d'un niveau de représentation à l'autre, dans un souci de simplification – et donc d'*utilisabilité* – de l'automate concerné.

Or, il se fait que plusieurs DFAs peuvent également être *équivalents* dans le sens où ils définissent le même langage. La Figure 2.8 montre un exemple de quatre DFAs équivalents, qui définissent le langage « (a ou b) suivi de (b ou c) ». Parmi ces DFAs, il en existe un qui compte un *nombre minimum d'états* et que l'on qualifie de *minimal*. Dans notre exemple, il s'agit de l'automate (4). Cet automate minimal est très intéressant, parce qu'il *minimise la place requise* pour le représenter.

Dans cette section, nous commençons par expliquer comment tester l'équivalence des états d'un DFA.

Une conséquence importante de ce test est qu'il offre un moyen de *minimiser* un DFA. Ainsi, à partir d'un DFA donné, il est possible de trouver un DFA équivalent qui compte un nombre minimum d'états. Notons que le DFA minimal est *unique* par nature (Moore 1956) : en effet, étant donné deux DFAs minimaux qui sont équivalents, il y a toujours moyen de renommer leurs états de sorte que les deux DFAs soient identiques.

Une autre conséquence de ce test d'équivalence des états est qu'il permet de déterminer l'équivalence de deux DFAs.

2.4.1 Tester l'équivalence des états

Définition 2.4.1. *Etant donné un DFA $A_D = (\Sigma, Q, i, F, E)$, $\forall p$ et $q \in Q$, p et q sont dits* équivalents *si, $\forall u \in \Sigma^*$, $\hat{\delta}(p, u) \cap F \neq \emptyset$ si et seulement si $\hat{\delta}(q, u) \cap F \neq \emptyset$.*

Notons que la définition 2.4.1 n'impose pas que $\hat{\delta}(p, u)$ et $\hat{\delta}(q, u)$ donnent *le même état*, mais simplement que les états atteints soient tous les deux finaux ou non finaux.

Require: Un ϵ-NFA $A_\epsilon = (\Sigma, Q, i, F, E)$
Ensure: Le NFA $A_N = (\Sigma, Q', i', F', E')$ équivalent à A_ϵ
/* étape 1 : clôture-ϵ */
$Q_\epsilon[p'] \leftarrow \emptyset$ **for** $p' \leftarrow 0$ **to** $|Q| - 1$
`CLOTURE-EPS`(Q_ϵ, p') **for** $p' \leftarrow 0$ **to** $|Q| - 1$
/* étape 2 : suppression-ϵ */
$H \leftarrow F' \leftarrow E' \leftarrow \emptyset$
$p' \leftarrow i' \leftarrow$ `INSERT`$(H, Q_\epsilon[i])$ /* 1^{er} élément $\rightarrow$ retour $= 0$ */
$Q' \leftarrow \{i'\}$
while $p' < |Q'|$ **do**
  $S_\epsilon \leftarrow$ `GET`(H, p')
  if $F \cap S_\epsilon \neq \emptyset$ **then**
    $F' \leftarrow F' \cup \{p'\}$
  end if
  for each $p \in S_\epsilon$ **do**
    for each $e \in E[p] : e.a \neq \epsilon$ **do**
      $q' \leftarrow$ `INSERT`$(H, Q_\epsilon[e.q])$
      $Q' \leftarrow Q' \cup \{q'\}$
      $E' \leftarrow E' \cup \{(p', e.a, q')\}$
    end for
  end for
  $p' = p' + 1$
end while
$A_N \leftarrow (\Sigma, Q', i', F', E')$
return A_N

Pseudocode 3: `SUPPRESSION-EPS`

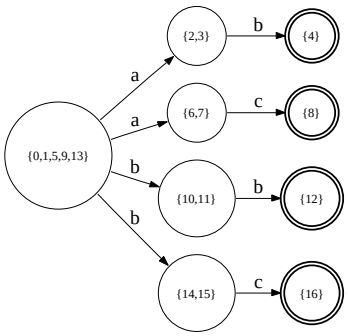


FIG. 2.7: Automate obtenu par suppression-ϵ de A_ϵ

Définition 2.4.2. *Si deux états p et q ne sont pas équivalents, ils sont dits* distinguables*, ce qui signifie qu'il existe au moins une string u tel que si $\hat{\delta}(p,u)$ ou $\hat{\delta}(q,u)$ est final, l'autre est non final.*

La façon la plus aisée de déterminer les états d'un DFA A_D qui sont équivalents est de commencer par déterminer les états de A_D qui sont distinguables. L'idée est de découvrir *de manière itérative* toutes les *paires d'états* qui sont distinguables. La recherche de ces paires s'arrête dès qu'il n'est plus possible de distinguer de nouvelles paires d'une itération à l'autre. Les états qui ne sont pas distinguables sont dès lors équivalents.

Cet algorithme repose sur une *table de distinction* que l'on remplit récursivement. La Table 2.1 montre le résultat du remplissage pour l'automate 1 de la Figure 2.8. On y constate qu'après distinction de toutes les paires possibles (marquées d'une croix rouge), on peut dégager les paires d'états équivalents suivantes : $\{1,4\}$, $\{2,3\}$, $\{2,5\}$, $\{2,6\}$, $\{3,5\}$, $\{3,6\}$ et $\{5,6\}$.

Base. Intuitivement, on peut considérer que chaque itération de l'algorithme de distinction augmente de 1 la longueur des strings u testées dans la fonction $\hat{\delta}$, en commençant à 0. L'algorithme est donc initialisé avec $u = \epsilon$, ce qui revient à considérer que toute paire d'états p et q est distinguable si l'un des états est final et l'autre pas. La table de distinction est donc initialisée en marquant comme distinguables toutes les paires d'états dont seul un état est final.

Induction. Posons p et q, deux états tels que pour un symbole a donné, $\delta(p,a) = r$ et $\delta(q,a) = s$. Si r et s sont distinguables, il en est de même pour p et q. Ceci est justifié parce qu'il y a au moins une string u qui distingue r de s de sorte que au distingue p de q : en effet, $\{\hat{\delta}(r,u), \hat{\delta}(s,u)\}$ correspond à la même paire d'états que $\{\hat{\delta}(p,au), \hat{\delta}(q,au)\}$. Ce principe est applicable de manière itérative dès que la distinction entre les états finaux et non finaux est réalisée, et s'interrompt dès qu'aucune paire supplémentaire n'a pu être distinguée entre deux itérations successives. Les états non distinguables sont alors équivalents.

2.4.2 De l'équivalence à la minimisation

Une conséquence importante du test d'équivalence des états est que l'on peut minimiser les DFAs. Ceci signifie que, à partir d'un DFA donné, on peut trouver le DFA qui accepte le même langage et contient le nombre minimum d'états. En outre, si l'on exclut le mode de nommage choisi pour les états, ce DFA minimal est unique pour le langage considéré.

Définition 2.4.3. *Une partition d'un ensemble Q est un ensemble $\mathcal{P}$ de sous-ensembles non vides de Q deux à deux disjoints et qui forment un recouvrement de Q. Autrement dit, $\mathcal{P}$ est une partition de Q si et seulement si les parties de $\mathcal{P}$ sont non vides et que tout élément $q \in Q$ se trouve dans exactement l'une de ces parties.*

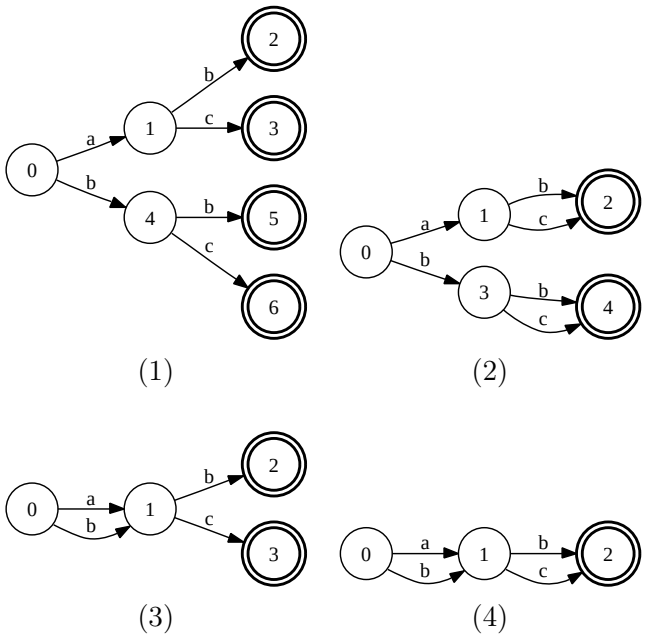


FIG. 2.8: Exemple de DFAs équivalents

1	×					
2	×	×				
3	×	×	v			
4	×	v	×	×		
5	×	×	v	v	×	
6	×	×	v	v	×	v
	0	1	2	3	4	5

TAB. 2.1: Table de distinction des états du DFA (1) de la Figure 2.8

Principe. Pour un DFA donné $A_D = (\Sigma, Q, i, F, E)$, le principe de l'algorithme est le suivant :

1. Etablir les paires d'états équivalents de Q à l'aide de la table de distinction.
2. Rassembler les états en ensembles, de sorte que tous les états équivalents soient dans un même ensemble, et que tout ensemble ne contienne que des états équivalents. Il s'agit donc de construire une *partition* $\mathcal{P}$ de Q.
3. Construire le DFA minimal équivalent $A_{min} = (\Sigma, Q', i', F', d')$ tel que :
   - Q' est la partition $\mathcal{P}$ de Q.
   - i' est l'ensemble $S \in Q'$ qui contient i.
   - F' contient tout ensemble $S \in Q'$ tel que $S \cap F \neq \emptyset$.
   - Etant donné les ensembles S et $T \in Q'$, et le symbole $a \in \Sigma$, $d'(S, a) = T$ puisque, pour tout état $s \in S$, $\delta(s, a) \in T$.

Appliqué à l'automate (1) de la Figure 2.8, la minimisation donne l'automate de la Figure 2.9, identique à l'automate (4) de la Figure 2.8.

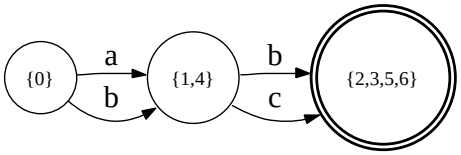


FIG. 2.9: Automate minimal A_{min}

Du principe à l'algorithme. L'algorithme de minimisation présenté en Pseudocode 4 s'écarte légèrement du principe détaillé ci-dessus, qui ne tient nullement compte du nombre gigantesque d'états que certains DFAs possèdent. Or, la taille de la table de distinction dépend directement du nombre d'états du DFA : pour un nombre n d'états, la table sera de taille $\frac{(n*(n-1))}{2}$, ce qui devient rapidement non représentable malgré la taille des mémoires des ordinateurs actuels.

L'idée sur laquelle repose l'algorithme proposé est de construire directement l'ensemble $\mathcal{P}$, et de le raffiner récursivement sans passer par la table de distinction. L'initialisation de $\mathcal{P}$ se fait en rassemblant tous les états non finaux dans un premier ensemble, et tous les états finaux dans un second (Pseudocode 5).

C'est sur cette base que travaille l'algorithme de partitionnement (Pseudocode 6), qui va récursivement raffiner $\mathcal{P}$ en tâchant de distinguer les états d'un même ensemble. Le principe de l'algorithme de partitionnement est de comparer, pour un ensemble S donné, un état s de S à tous les autres états de S ; tout état distinguable de s est déplacé dans un nouvel ensemble T, de sorte que S ne contient plus que les états que l'itération en cours n'a pas pu distinguer.

La distinction de deux états p et q (Pseudocode 7) se réalise toujours en comparant les états atteints par leurs transitions. Cependant, étant donné un symbole a tel que $\delta(p, a) = r$ et $\delta(q, a) = s$, r et s sont cette fois considérés comme distinguables s'ils appartiennent à des *ensembles différents* de $\mathcal{P}$. La construction du DFA minimal se déroule ensuite de manière tout à fait classique (Pseudocode 8).

Dans les différents Pseudocodes, nous employons la notation suivante :

- $P =(V_S, V_L)$ Cette structure gère une partition de l'ensemble Q. Les éléments de base de cette structure sont de type *liste chaînée* : $el =(state, set, prev, next)$: un état ($state$) connaît l'ensemble d'états équivalents (set) auquel il appartient, et connaît son prédécesseur ($prev$) et son successeur ($next$) dans l'ensemble. Les deux membres de P donnent des méthodes d'accès différentes à ces éléments :
  - V_S est un vecteur indexé des états : $V_S[i] = el : el.state = i$. Ce vecteur donne donc un accès direct aux *états*.
  - V_L est un vecteur indexé d'*ensembles* représentés sous la forme de listes chaînées. Une liste i donnée est accessible *via* son premier élément : $V_L[i].first = el : el.set = i$. Les autres éléments de la liste sont accessibles *via* les membres $prev$ et $next$. Le premier élément de la liste n'a pas de prédécesseur ($el.prev = 0$), et le dernier élément de la liste n'a pas de successeur ($el.next = 0$). Sur une liste sont définies les méthodes `INSERT` et `UPDATE`. La méthode `INSERT` permet d'insérer un élément *el en tête* d'une liste i donnée, mais ne met pas à jour la valeur de $el.set$. La méthode `UPDATE` permet de mettre à jour, pour une liste i donnée, le membre set de tous ses éléments de sorte que pour tout élément de la liste i, $el.set = i$.
  - Sur un élément el est définie la méthode `UNLINK`, qui permet de délier un élément el de son successeur et de son prédécesseur. Si le successeur et le prédécesseur existent, ceux-ci sont liés ensemble. La méthode retourne toujours le successeur de l'élément délié.
- $|S|$, S étant une liste ou un ensemble, signifie *nombre d'éléments* de S.

2.4.3 Tester l'équivalence de DFAs

Outre la minimisation, la notion d'équivalence des états permet également de tester l'équivalence de deux DFAs A_{D1} et A_{D2}. Pour ce faire, l'algorithme, présenté en Pseudocode 9, commence par réunir A_{D1} et A_{D2} dans un seul DFA A_{D3}, moyennant le renommage des états et des transitions de l'un des DFAs afin d'éviter toute ambiguïté (Pseudocode 10). Ceci étant réalisé, l'algorithme de partitionnement décrit précédemment est appliqué sur A_{D3}. La dernière étape de l'algorithme consiste simplement à vérifier que les états initiaux de A_{D1} et de A_{D2} appartiennent au même sous-ensemble S de la partition $\mathcal{P}$ de A_{D3}. Si c'est le cas, A_{D1} et A_{D2} sont équivalents.

Require: Un DFA $A_D = (\Sigma, Q, i, F, E)$
Ensure: Le DFA minimal $A_{min} = (\Sigma, Q', i', F', E')$ équivalent à A_D
1: $P \leftarrow$ `PARTITION`(A_D)
2: $A_{min} \leftarrow$ `PARTITION2FSM`(A_D, P)
3: **return** A_{min}

Pseudocode 4: `MINIMISATION`

Require: Un DFA $A_D = (\Sigma, Q, i, F, E)$
Ensure: $P = (V_S, V_L)$, partition initiale de Q, contient 2 sous-ensembles si $Q \neq F$, sinon un seul ;
i, l'état initial de Q, se trouve toujours dans le première liste
1: $initialFinal \leftarrow ((i \in F)$? `TRUE` : `FALSE`$)$
2: $V_S \leftarrow V_L \leftarrow \emptyset$
3: **for** $p' \leftarrow 0$ to $|Q| - 1$ **do**
4: $el \leftarrow (p', 0, 0, 0)$
5: $V_S[p'] \leftarrow el$
6: `INSERT`$(V_L[0], el)$
7: **end for**
8: $el \leftarrow V_L[0].first$
9: **while** $el \neq 0$ **do**
10: **if** $(el.state \in F) = initialFinal$ **then**
11: $el \leftarrow el.next$
12: **else**
13: $el_n \leftarrow$ `UNLINK`(el)
14: `INSERT`$(V_L[1], el)$
15: $el \leftarrow el_n$
16: **end if**
17: **end while**
18: **if** $|V_L| = 2$ **then**
19: `UPDATE`$(V_L, 1)$
20: **end if**
21: $P \leftarrow (V_S, V_L)$
22: **return** P

Pseudocode 5: `PARTITION-FINAL`

Require: Un DFA $A_D = (\Sigma, Q, i, F, E)$
Ensure: $P = (V_S, V_L)$ contient la partition de Q tel que chaque sous-ensemble S de P contient des états équivalents et que tous les états équivalents sont dans le même sous-ensemble

1: $P \leftarrow$ `PARTITION-FINAL`(A_D)
2: $size_c \leftarrow 0$
3: $size_t \leftarrow |P.V_L|$
4: **while** $size_c < size_t$ **do**
5: $size_c \leftarrow size_t$
6: $j \leftarrow 0$
7: **while** $j < |P.V_L|$ **do**
8: $size_t \leftarrow |P.V_L|$
9: $el_1 \leftarrow P.V_L[j].first$
10: $el_c \leftarrow el_1.next$
11: **while** $el_c \neq 0$ **do**
12: $dist \leftarrow$ `DISTINGUABLE`(A_D, P, el_1, el_c)
13: **if** $dist =$ `TRUE` **then**
14: $k \leftarrow size_t$
15: **while** $dist =$ `TRUE` **and** $k < |P.V_L|$ **do**
16: $el_{1'} \leftarrow P.V_L[k].first$
17: **if** $|E[el_c.state]| = |E[el_{1'}.state]|$ **then**
18: $dist \leftarrow$ `FALSE`
19: **else**
20: $k \leftarrow k + 1$
21: **end if**
22: **end while**
23: $el_n \leftarrow$ `UNLINK`(el_c)
24: `INSERT`$(P.V_L[k], el_c)$
25: $el_c \leftarrow el_n$
26: **else**
27: $el_c \leftarrow el_c.next$
28: **end if**
29: **end while**
30: $j \leftarrow j + 1$
31: `UPDATE`$(P.V_L, k)$ **for** $k \leftarrow size_t$ **to** $|P.V_L|$
32: **end while**
33: $size_t \leftarrow |P.V_L|$
34: **end while**
35: **return** P

Pseudocode 6: `PARTITION`

Require: Un DFA $A_D = (\Sigma, Q, i, F, E)$;
$P = (V_S, V_L)$, partition de Q, au moins initialisée par `PARTITION-FINAL` ;
el_1 et el_2, 2 éléments de type $el = (state, set, prev, next)$
Ensure: retourne `TRUE` si les états $el_1.state$ et $el_2.state$ sont *distinguables*, `FALSE` sinon

```
1: dist ← TRUE
2: if |E[el1.state]| = |E[el2.state]| then
3:    dist ← FALSE
4:    i ← 0
5:    while dist = FALSE and i < |E[el1.state]| do
6:       e1 ← E[el1.state][i]
7:       e2 ← E[el2.state][i]
8:       if e1.a ≠ e2.a or P.VS[e1.q].set ≠ P.VS[e2.q].set then
9:          dist ← TRUE
10:      else
11:         i ← i + 1
12:      end if
13:   end while
14: end if
15: return dist
```

Pseudocode 7: `DISTINGUABLE`

Require: Un DFA $A_D = (\Sigma, Q, i, F, E)$;
$P = (V_S, V_L)$ contient une partition de Q
Ensure: Le DFA minimal $A_{min} = (\Sigma, Q', i', F', E')$ équivalent à A_D

```
1: Q' ← F' ← E' ← ∅
2: i' ← 0
3: for p' ← 0 to |P.VL| do
4:    elc ← P.VL[p'].first
5:    Q' ← Q' ∪ {p'}
6:    if elc.state ∈ F then
7:       F' ← F' ∪ {p'}
8:    end if
9:    for each e ∈ Ec do
10:      E' ← E' ∪ {(p', e.a, P.VS[e.q].set)}
11:   end for
12: end for
13: Amin ←(Σ, Q', i', F', E')
14: return Amin
```

Pseudocode 8: `PARTITION2FSM`

Require: Deux DFAs $A_{D1} = (\Sigma, Q_1, i_1, F_1, E_1)$ et $A_{D2} = (\Sigma, Q_2, i_2, F_2, E_2)$
Ensure: retourne `TRUE` si A_{D1} et A_{D2} sont *équivalents*, `FALSE` sinon
1: $A_{D3} \leftarrow$ `COMBINE`(A_{D1}, A_{D2})
2: $i'_2 \leftarrow i_2 + |Q_1|$
3: $P \leftarrow$ `PARTITION`(A_{D3})
4: **if** $P.V_S[i_1].set = P.V_S[i'_2].set$ **then**
5: **return** `TRUE`
6: **end if**
7: **return** `FALSE`

Pseudocode 9: `EQUIVALENT`

Require: Deux DFAs $A_{D1} = (\Sigma, Q_1, i_1, F_1, E_1)$ et $A_{D2} = (\Sigma, Q_2, i_2, F_2, E_2)$
Ensure: Le DFA $A_{D3} = (\Sigma, Q_3, i_1, F_3, E_3)$ combine A_{D1} et A_{D2}
1: $Q_3 \leftarrow Q_1$
2: $F_3 \leftarrow F_1$
3: **for each** $p \in Q_2$ **do**
4: $Q_3 \leftarrow Q_3 \cup \{p + |Q_1|\}$
5: $F_3 \leftarrow F_3 \cup \{p + |Q_1|\}$ **if** $p \in F_2$
6: **end for**
7: $E_3 \leftarrow E_1$
8: **for each** $e \in E_2$ **do**
9: $E_3 \leftarrow E_3 \cup \{(e.p + |Q_1|, e.a, e.q + |Q_1|)\}$
10: **end for**
11: $A_{D3} \leftarrow (\Sigma, Q_3, i_1, F_3, E_3)$
12: **return** A_{D3}

Pseudocode 10: `COMBINE`

2.5 Opérations régulières

La puissance des automates a été mise en évidence par Kleene (1956), qui les relie à la classe des langages réguliers en démontrant que le monoïde libre Σ^* est clos pour une série d'opérations. Ces opérations sont les *propriétés de clôture* des langages réguliers.

Note 2.5.1. On dit d'un ensemble qu'il est clos pour une opération lorsque cette opération, appliquée à des éléments de l'ensemble, donne un élément appartenant lui-même à l'ensemble.

2.5.1 Théorème de Kleene

Théorème 2.5.1 (Kleene, 1956)**.** *La famille des langages réguliers sur Σ^* est égale à la plus petite famille de langages sur Σ^* qui contient l'ensemble vide, les ensembles singleton et est close sous la concaténation, l'union et l'étoile de Kleene.*

Clôture sous la concaténation. Etant donné deux automates A_1 et A_2, il est possible de calculer un automate $A_1 \cdot A_2$ de sorte que $L(A_1 \cdot A_2) = L(A_1) \cdot L(A_2)$.

Clôture sous l'union. Etant donné deux automates A_1 et A_2, il est possible de calculer un automate $A_1 \cup A_2$ de sorte que $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$.

Clôture sous l'étoile de Kleene. Etant donné l'automate A, il est possible de calculer un automate A^* de sorte que $L(A^*) = L(A)^*$.

Etant donné un alphabet $\Sigma = \{a, b, c\}$ par exemple, la famille des langages réguliers sur Σ^* se construit à partir de l'ensemble vide $\emptyset$ et des ensembles singleton $(\{a\}, \{b\}, \{c\})$ par application récursive de la concaténation, de l'union et/ou de l'étoile de Kleene : $\{\epsilon$, a, ab, ba, $\ldots$, caa, $\ldots$, $cccbabbc$, $\ldots\}$.

En assimilant les automates aux langages réguliers, ce théorème démontre que les automates sont équivalents aux *expressions régulières*. Sans entrer à ce stade dans une explication détaillée, voici un exemple d'expression régulière

$$\texttt{[A-Z][a-z]}^* \tag{2.5.1.1}$$

qui signifie « une lettre majuscule comprise entre A et Z, suivie par zéro, une ou plusieurs lettres minuscules comprises entre a et z ». Les expressions régulières sont donc des descriptions syntaxiques. Elles sont utilisées dans de nombreux langages amenés à traiter des chaînes de caractères. On peut citer par exemple :

1. Les commandes de recherche sous UNIX, comme `grep`, qui permettent de rechercher des chaînes de caractères.
2. Les analyseurs lexicaux, comme Lex, qui sont utilisés au premier stade de la compilation de programmes afin d'identifier les *unités lexicales* d'un programme : mot-clé, variable, constante, fonction, etc.

En somme, les automates sont équivalents à des *descriptions syntaxiques*. Il est dès lors possible de décrire de manière concise le langage d'un automate sous la forme d'une expression régulière.

La syntaxe des expressions régulières et le calcul de l'automate correspondant à une expression donnée sont décrits en Section 5.2.

2.5.2 Autres propriétés de clôture

Dans la continuité du théorème de Kleene, il a été démontré que les automates sont également clos sous l'intersection et la complémentation.

Clôture sous l'intersection. Etant donné deux automates $A_1 = (\Sigma, Q_1, i_1, F_1, E_1)$ et $A_2 = (\Sigma, Q_2, i_2, F_2, E_2)$, il est possible de calculer un automate $A_1 \cap A_2$ de sorte que $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$. Cet automate correspond au produit cartésien de A_1 et A_2 : $A_1 \cap A_2 = (\Sigma, Q_1 \times Q_2, i_1 \times i_2, F_1 \times F_2, E)$ avec :

$$E = \bigcup_{(p_1,a,q_1)\in E_1,(p_2,a,q_2)\in E_2} (\{p_1, p_2\}, a, \{q_1, q_2\}) \tag{2.5.2.1}$$

Le Pseudocode 11 présente un algorithme simple calculant l'intersection de deux automates. Dans le Pseudocode, nous employons la notation suivante :

- $E[p]$ désigne le sous-ensemble de E qui ne contient que les transitions quittant l'état p.
- e est une transition de type (p, a, q) dont $e.p$ identifie l'état courant, $e.q$, l'état suivant et $e.a$, le symbole.
- H est une structure qui permet d'insérer (`INSERT`) et de récupérer (`GET`) des éléments de type *ensemble*. La méthode `INSERT` insère une et une seule fois un élément donné, mais en retourne toujours l'indice d'insertion. Cet indice est initialisé à 0 et est incrémenté de 1 *après* l'insertion d'un nouvel élément.
- $|S|$, où S est un ensemble, signifie *nombre d'éléments* de S.

Clôture sous la complémentation. Etant donné un automate A, il est possible de calculer un automate $\bar{A}$ de sorte que $L(\bar{A}) = \Sigma^* - L(A)$. Le complément d'un automate A représente donc tout ce qui, dans Σ^*, n'est pas A.

La construction du complément d'un automate $A = (\Sigma, Q, i, F, d)$ est fort simple, pour autant que A soit déterministe. Le Pseudocode 12 en présente l'algorithme. Le complément de A est $\bar{A} = (\Sigma, Q', i, F'd')$. Q' et d' sont initialisés comme des copies respectivement de Q et de d (lignes 1–2). F' est initialisé comme l'ensemble des états non finaux de Q (ligne 3). Si A n'est pas complet (cf. Définition 2.3.1), $\bar{A}$ est rendu complet. Pour ce faire, un nouvel état complet q est ajouté à Q' et à F' (lignes 13–14). Cet état q est complet parce

qu'il existe, pour tout symbole a de Σ, exactement une transition (q, a, q) (ligne 15). Enfin, pour tout état p de Q' qui est incomplet, une transition (p, a, q) par symbole a manquant est créée (lignes 6–11). La Figure 2.10 montre un automate et le résultat du calcul de son complément.

En pratique, intersection et complémentation permettent, par exemple, de définir des contraintes sous la forme d'automates. Afin d'appliquer une contrainte C, son complément $\bar{C}$ sera préalablement calculé. L'application de la contrainte à un langage L correspondra ensuite à la construction de l'intersection entre $\bar{C}$ et L : $L_C = \bar{C} \cap L$.

La Figure 2.11 donne un petit exemple de contrainte syntaxique. Le premier automate est la définition de la contrainte (C), qui interdit qu'un pronom personnel complément direct (`PRONPERCD`) soit suivi d'un nom (`NOUN`). Le second automate est le complément de la contrainte ($\bar{C}$), qui sera appliqué sur le langage par intersection.

Ceci fait des automates des outils très puissants, étant donné que d'autres formalismes, comme les grammaires hors-contexte (cf. Section 1.3.2), sont clos sous la concaténation, l'union et l'étoile de Kleene, mais pas sous l'intersection ni la complémentation.

2.6 Synthèse

Les automates à états finis, vus comme des graphes orientés étiquetés, acceptent différents niveaux de représentation équivalents : du ϵ-NFA qui est le résultat de certaines opérations sur les FSMs, au DFA dont le temps de parcours est linéairement proportionnel à la taille de l'entrée à analyser. Dans un souci d'efficacité, des algorithmes ont été définis afin de supprimer les transitions ϵ d'un ϵ-NFA, et de déterminiser un NFA. En outre, un DFA peut toujours être minimisé de manière à réduire la place requise pour le représenter.

Enfin et peut-être surtout, les automates à états finis définissent des langages réguliers. Ils sont de ce fait clos sous les opérations régulières définies sur ces langages : union, concaténation et étoile de Kleene comme les automates à pile, mais également intersection et complémentation au contraire des automates à pile. Cette équivalence entre automates à états finis et langages réguliers rend les automates équivalents aux expressions régulières, descriptions syntaxiques qui permettent d'exprimer en compréhension ce que les automates représentent en extension.

Les automates sont donc des outils puissants, qui constituent la base solide d'autres machines à états finis performantes : les transducteurs et les machines pondérées.

Require: Deux FSAs $A_1 = (\Sigma, Q_1, i_1, F_1, E_1)$ et $A_2 = (\Sigma, Q_2, i_2, F_2, E_2)$
Ensure: $A_3 = (\Sigma, Q_3, i_3, F_3, E_3) = A_1 \cap A_2$
1: $A_3 \leftarrow 0$
2: $H \leftarrow F_3 \leftarrow E_3 \leftarrow \emptyset$
3: $p_3 \leftarrow i_3 \leftarrow$ `INSERT`$(H, (i_1, i_2))$ /* 1^{er} élément $\rightarrow$ retour $= 0$ */
4: $Q_3 \leftarrow \{i_3\}$
5: **while** $p_3 < |Q_3|$ **do**
6: $(p_1, p_2) \leftarrow$ `GET`(H, p_3)
7: $F_3 \leftarrow F_3 \cup \{p_3\}$ **if** $p_1 \in F_1$ **and** $p_2 \in F_2$
8: **for each** $(e_1, e_2) : e_1 \in E_1[p_1], e_2 \in E_2[p_2], e_1.a = e_2.a$ **do**
9: $q_3 \leftarrow$ `INSERT`$(H, (e_1.q, e_2.q))$
10: $Q_3 \leftarrow Q_3 \cup \{q_3\}$
11: $E_3 \leftarrow E_3 \cup \{(p_3, e_1.a, q_3)\}$
12: **end for**
13: $p_3 = p_3 + 1$
14: **end while**
15: $A_3 \leftarrow (\Sigma, Q_3, i_3, F_3, E_3)$ **if** $E_3 \neq \emptyset$
16: **return** A_3

Pseudocode 11: `INTERSECTION`

Require: Un DFA $A_D = (\Sigma, Q, i, F, d)$
Ensure: Le DFA complet $\bar{A_D} = (\Sigma, Q', i, F', d')$ complément de A_D
1: $Q' \leftarrow Q$
2: $d' \leftarrow d$
3: $F' \leftarrow (Q - F)$
4: $q \leftarrow |Q'|$
5: *complete* $\leftarrow$ `TRUE`
6: **for each** $p \in Q'$ **do**
7: **for each** $a \in \Sigma : d'(p, a) = \emptyset$ **do**
8: $d'(p, a) \leftarrow q$
9: *complete* $\leftarrow$ `FALSE`
10: **end for**
11: **end for**
12: **if** *complete* $=$ `FALSE` **then**
13: $Q' \leftarrow Q' \cup q$
14: $F' \leftarrow F' \cup q$
15: $d'(q, a) \leftarrow q$ **for each** $a \in \Sigma$
16: **end if**
17: $\bar{A_D} \leftarrow (\Sigma, Q', i, F', E')$
18: **return** $\bar{A_D}$

Pseudocode 12: `COMPLEMENT`

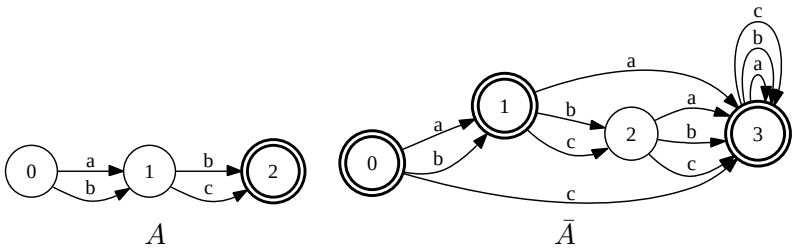
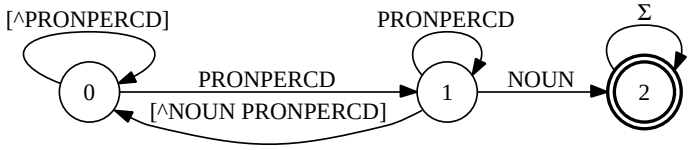
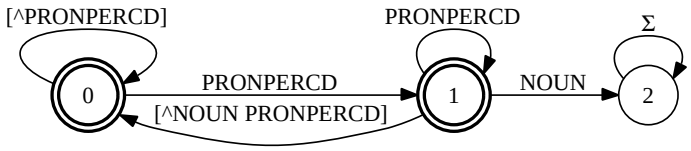


FIG. 2.10: Un automate et son complément définis sur $\Sigma = \{a, b, c\}$



1. C



2. $\bar{C}$

FIG. 2.11: Contrainte exprimée sous la forme d'un automate

Chapitre 3

Les transducteurs

3.1 Introduction

Les transducteurs peuvent être considérés comme une classe de graphes, mais également comme une classe de relations ou de transductions entre strings. Comme dans le cas des automates, les transducteurs sont clos sous certaines opérations. Parmi celles-ci, une est fondamentale : la composition, qui autorise la construction de relations complexes à partir de relations simples. Cette opération fait des transducteurs des outils puissants et essentiels dans la domaine des machines à états finis. Il existe en outre une classe particulière de transducteurs : les transducteurs séquentiels qui, comme les automates déterministes, assurent un temps de parcours linéairement proportionnel à la string d'entrée et permettent un niveau de minimisation optimisé.

Après avoir rappelé les principales définitions relatives aux transducteurs, cette section présente les opérations régulières qui leur sont applicables. La section se termine avec la description des transducteurs séquentiels et des algorithmes d'optimisation définis sur ces transducteurs.

3.2 Définitions

Les transducteurs peuvent être considérés comme définissant une classe de graphes, une classe de relations entre strings ou une classe de transductions entre strings.

3.2.1 Graphes

Selon la première interprétation, les transducteurs peuvent être vus comme des automates dont chaque arc est étiqueté par une paire de symboles plutôt que par un seul symbole.

Définition 3.2.1. *Un transducteur à états finis T est un 6-uplet* $(\Sigma_1, \Sigma_2, Q, i,$
$F, E)$ *où* Σ_1 *est un ensemble fini de symboles appelé alphabet d'entrée,* Σ_2 *est un ensemble*

fini de symboles appelé alphabet de sortie, Q est un ensemble fini d'états, $i \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux, et $E \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ est un ensemble fini de transitions de la forme (p, a_1, a_2, q).

Remarquons que l'ensemble E peut être remplacé par une fonction de transition δ qui projette $Q \times \Sigma_1^*$ sur $\mathcal{P}^Q$, et une fonction d'émission σ qui projette $Q \times \Sigma_1^*$ sur $\mathcal{P}^{\Sigma_2^*}$. Il y a équivalence des deux définitions :

$$\begin{aligned} \delta(p, a_1) &= \{q \in Q \mid \exists (p, a_1, a_2, q) \in E\} \\ \sigma(p, a_1) &= \{a_2 \in \Sigma_2^* \mid \exists (p, a_1, a_2, q) \in E\} \end{aligned} \tag{3.2.1.1}$$

Un transducteur peut toujours être considéré comme un automate, pour autant que l'on définisse la notion d'*automate sous-jacent*.

Définition 3.2.2. *Si $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ est un transducteur, son* automate sous-jacent *$A = (\Sigma, Q, i, F, E')$ est tel que :*

$$\begin{aligned} &\Sigma = \Sigma_1 \times \Sigma_2 \\ &(p, (a_1, a_2), q) \in E' \textit{ ssi } (p, a_1, a_2, q) \in E \end{aligned} \tag{3.2.1.2}$$

Toutes les propriétés définies sur les automates, telles que la *déterminisation* et la *minimisation*, sont applicables à l'automate sous-jacent d'un transducteur, ce qui peut s'avérer très intéressant. Notons cependant que la déterminisation d'un transducteur au travers de son automate sous-jacent n'assure pas pour autant un temps de calcul linéairement proportionnel à la taille de *l'entrée* à analyser, contrairement à l'algorithme défini sur les transducteurs séquentiels. Nous revenons sur ce point dans la Section 3.4.

A l'inverse, un automate peut toujours être converti en transducteur. On parle de *transducteur identitaire*.

Définition 3.2.3. *Un* transducteur identitaire *est un transducteur obtenu à partir d'un automate auquel on a ajouté des étiquettes de sortie identiques aux étiquettes d'entrée. Plus formellement, Si $A = (\Sigma, Q, i, F, E)$ est un automate, son* transducteur identitaire *$T = (\Sigma, \Sigma, Q, i, F, E')$ est tel que :*

$$(p, a, a, q) \in E' \textit{ ssi } (p, a, q) \in E \tag{3.2.1.3}$$

On notera que la Définition 3.2.1 présente un ensemble de transitions E qui autorise les transducteurs à travailler sur des strings (*e.g.*, $(p, ab, cdef, q)$) et sur la transition vide $(p, \epsilon, \epsilon, q)$. Pour le bon déroulement de certains algorithmes, il peut être nécessaire de définir un transducteur dont les transitions ne travaillent que sur des symboles. La littérature anglophone parle de *letter transducer*, ce que nous traduisons par *transducteur lettre-à-lettre*.

Définition 3.2.4. *Un transducteur* $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ *est un* transducteur lettre-à-lettre *si et seulement si :*
(a) $E \cap (Q \times \{\epsilon\} \times \{\epsilon\} \times Q) = \emptyset$
(b) $E \subseteq (Q \times (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times Q)$

Etant donné un transducteur $T_1 = (\Sigma_1, \Sigma_2, Q_1, i_1, F_1, E_1)$, il est toujours possible de calculer le transducteur lettre-à-lettre $T2 = (\Sigma_1, \Sigma_2, Q_2, i_2, F_2, E_2)$, de sorte que $|T_1| = |T_2|$. Ceci implique que :
(a) $E_1 \cap (Q \times \{\epsilon\} \times \{\epsilon\} \times Q) = \emptyset$
(b) $E_2 \subseteq (Q_1 \times (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times Q_2)$

Informellement, (a) est obtenu en éliminant les transitions ϵ de E_1 à l'aide de l'algorithme suppression-ϵ, à condition de considérer l'automate sous-jacent de T_1 dans lequel (ϵ, ϵ) est la transition ϵ. (b) s'obtient en créant des transitions lettre-à-lettre à partir des transitions de T_1, simplement en ajoutant de nouveaux états intermédiaires dans T_2.

Lorsqu'un transducteur lettre-à-lettre ne présente aucun ϵ ni en entrée, ni en sortie de transition, on parle de *transducteur lettre-à-lettre sans* ϵ.

Définition 3.2.5. *Un transducteur* $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ *est un* transducteur lettre-à-lettre sans ϵ *si et seulement si* $E \subseteq (Q \times \Sigma_1 \times \Sigma_2 \times Q)$.

Il est également parfois bien utile de pouvoir ne considérer que les symboles d'entrée ou les symboles de sortie d'un transducteur, ce qui nous conduit à définir :

Définition 3.2.6. *La* première projection *et la* seconde projection *d'un transducteur* $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ *sont les automates* $A_{p1(T)}$ *et* $A_{p2(T)}$ *de sorte que :*

$$\begin{aligned} A_{p1(T)} &= (\Sigma_1, Q, i, F, E_{p1}) \\ &\textit{tel que } E_{p1} = \{(p, a_1, q) : (p, a_1, a_2, q) \in E\} \end{aligned} \tag{3.2.1.4}$$

$$\begin{aligned} A_{p2(T)} &= (\Sigma_2, Q, i, F, E_{p2}) \\ &\textit{tel que } E_{p2} = \{(p, a_2, q) : (p, a_1, a_2, q) \in E\} \end{aligned} \tag{3.2.1.5}$$

Notons que si un transducteur présente des transitions du type (p, a_1, ϵ, q) (resp. (p, ϵ, a_2, q)), sa première projection (resp. seconde projection) présentera des transitions ϵ qu'il sera utile de supprimer (cf. Section 2.3.3) afin d'éviter tout délai lors du parcours d'une string d'entrée.

3.2.2 Relations

Selon la deuxième interprétation, les transducteurs définissent des relations entre strings. Cette interprétation demande préalablement d'étendre l'ensemble de transitions E de sorte qu'il opère sur des strings :

Définition 3.2.7. *L'ensemble étendu de transitions* $\hat{E} \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ *est défini comme le plus petit ensemble tel que :*

(i) $\forall q \in Q, (q, \epsilon, \epsilon, q) \in \hat{E}$

(ii) $\forall u_1 \in \Sigma_1^*, \forall u_2 \in \Sigma_2^*,$ *et* $\forall a_1 \in \Sigma_1, \forall a_2 \in \Sigma_2,$
si $(p, u_1, u_2, q) \in \hat{E}$ *et* $(q, a_1, a_2, r) \in E,$ *alors* $(p, u_1a_1, u_2a_2, r) \in \hat{E}$

Ces extensions étant faites, un transducteur T peut maintenant être considéré comme définissant une relation $L(T)$ comme suit :

$$L(T) = \{(u_1, u_2) \in \Sigma_1^* \times \Sigma_2^* \mid \exists (i, u_1, u_2, q) \in \hat{E} \text{ avec } q \in F\} \tag{3.2.2.1}$$

La notion de projection posée en Définition 3.2.6 s'applique dès lors à une relation $L(T)$ comme suit :

Proposition 3.2.1. *Si* $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ *est un transducteur, alors*

$$L(A_{p1(T)}) = \{u_1 \in \Sigma_1^* \mid \exists u_2 \in \Sigma_2^* \text{ tel que } (u_1, u_2) \in L(T)\} \tag{3.2.2.2}$$
$$L(A_{p2(T)}) = \{u_2 \in \Sigma_2^* \mid \exists u_1 \in \Sigma_1^* \text{ tel que } (u_1, u_2) \in L(T)\} \tag{3.2.2.3}$$

3.2.3 Transductions

Selon la troisième interprétation, un transducteur T peut être vu comme une correspondance $|T|$ entre l'ensemble de strings Σ_1^* et $\mathcal{P}^{\Sigma_2^*}$, l'ensemble des parties de l'ensemble de strings Σ_2^* :

$$|T|(u_1) = \{u_2 \in \Sigma_2^* : (u_1, u_2) \in L(T)\} \tag{3.2.3.1}$$

Cette notation peut être étendue de manière à s'appliquer à des ensembles de strings :

$$\text{si } U_1 \subseteq \Sigma_1^*, \ |T|(U_1) = \bigcup_{u_1 \in U_1} |T|(u_1) \tag{3.2.3.2}$$

Ceci nous amène à définir la différence entre *transduction rationnelle* et *fonction rationnelle*.

Définition 3.2.8. *Une transduction* $\tau : \Sigma_1^* \to \mathcal{P}^{\Sigma_2^*}$ *est une* transduction rationnelle *s'il existe un transducteur* T *tel que* $\tau = |T|$. *Si, pour toute string* u_1 *de l'ensemble d'entrée* Σ_1^*, $|T|(u_1)$ *est soit l'ensemble vide, soit un singleton,* $|T|$ *est* une fonction rationnelle.

Ainsi, au contraire de la transduction rationnelle qui établit une correspondance entre Σ_1^* et $\mathcal{P}^{\Sigma_2^*}$, une fonction rationnelle établit une correspondance entre Σ_1^* et Σ_2^*.

3.3 Opérations régulières

La puissance des transducteurs, comme celle des automates, leur vient de leurs propriétés de clôture et de leurs propriétés algorithmiques. Tous les transducteurs sont clos sous l'union et l'inverse. Les transducteurs lettre-à-lettre sont également clos sous la composition, et les transducteurs lettre-à-lettre sans ϵ sont clos sous l'intersection.

3.3.1 Clôture sous l'union

Etant donné deux transducteurs T_1 et T_2, il est possible de calculer un transducteur $T_1 \cup T_2$ tel que $|T_1 \cup T_2| = |T_1| \cup |T_2|$:

$$\forall u_1 \in \Sigma_1^*,\ |T_1 \cup T_2|(u_1) = |T_1|(u_1) \cup |T_2|(u_1) \tag{3.3.1.1}$$

3.3.2 Clôture sous l'inverse

Etant donné un transducteur $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, il existe un transducteur T^{-1} tel que $|T^{-1}|(u_2) = \{u_1 \in \Sigma_1^* \mid u_2 \in |T|(u_1)\}$. Le transducteur $(\Sigma_2, \Sigma_1, Q, i, F, E^{-1})$ est ce transducteur, tel que

$$(p, a_2, a_1, q) \in E^{-1} \textit{ ssi } (p, a_1, a_2, q) \in E \tag{3.3.2.1}$$

3.3.3 Clôture sous la composition

La composition est en fait une généralisation de l'intersection des automates : étant donné deux transducteurs T_1 et T_2, la composition implique que le langage de sortie de T_1 et le langage d'entrée de T_2 soient définis sur le même alphabet, de manière à pouvoir calculer leur intersection. Si intersection il y a, la composition construit un transducteur T_3 qui met en relation le langage d'entrée de T_1 et le langage de sortie de T_2, réduits à l'intersection calculée. La composition est donc l'opération qui permet de combiner différents niveaux de représentation pour construire des relations complexes à partir de relations simples.

De manière plus formelle, étant donné deux transducteurs lettre-à-lettre $T_1 = (\Sigma_1, \Sigma_2, Q_1, i_1, F_1, E_1)$ et $T_2 = (\Sigma_2, \Sigma_3, Q_2, i_2, F_2, E_2)$, il existe un transducteur $T_1 \circ T_2$ tel que, pour chaque $u_1 \in \Sigma_1^*$, $|T_1 \circ T_2|(u_1) = |T_2|([T_1|(u_1))$. Le transducteur $T_3 = (\Sigma_1, \Sigma_3, Q_1 \times Q_2, (i_1, i_2), F_1 \times F_2, E_3)$ satisfait $|T_3|(u_1) = |T_1 \circ T_2|(u_1) = |T_2|([T_1|(u_1))$ tel que

$$\begin{aligned} E_3 = \quad & \{((p_1, p_2), a_1, a_3, (q_1, q_2)) : \exists a_2 \in \Sigma_2 \cup \{\epsilon\} \text{ tel que} \\ & (p_1, a_1, a_2, q_1) \in Q_1 \textit{ et } (p_2, a_2, a_3, q_2) \in Q_2\} \end{aligned} \tag{3.3.3.1}$$

Algorithme. L'algorithme qui réalise la composition de deux transducteurs lettre-à-lettre sans ϵ est évidemment fort proche de l'intersection présentée en Pseudocode 11, moyennant les modifications suivantes :

11 : **for each** $(e_1, e_2) : e_1 \in E_1[p_1], e_2 \in E_2[p_2], e_1.a_2 = e_2.a_1$ **do**
12 : $q_3 \leftarrow \texttt{INSERT}(H, (e_1.q, e_2.q))$
13 : $Q_3 \leftarrow Q_3 \cup \{q_3\}$
14 : $E_3 \leftarrow E_3 \cup \{(p_3, e_1.a_1, e_2.a_2, q_3)\}$
15 : **end for**

Dans ce Pseudocode, e est une transition de type (p, a_1, a_2, q) dont $e.a_1$ identifie le symbole d'entrée et $e.a_2$, le symbole de sortie.

L'opération se complique cependant lorsque les transducteurs peuvent être étiquetés par des transitions ϵ en entrée et/ou en sortie, étant donné qu'une transition ϵ autorise – mais n'oblige pas – à avancer dans le transducteur correspondant sans spécialement lire de symbole dans l'autre transducteur. Les transitions ϵ multiplient donc les chemins possibles dans la composition, comme l'illustre la Figure 3.1.

Or, l'idéal est de ne proposer qu'un seul chemin, que l'on pourrait considérer comme *le meilleur alignement* entre les deux transducteurs.

Pereira & Riley (1997) ont proposé une méthode qui permet de choisir, *au cours de la composition*, l'unique chemin à conserver dans le résultat. Le principe est de placer entre les deux transducteurs à composer un autre transducteur F qui sert de *filtre*, et n'autorise que certaines suites de transitions :

$$T_1 \circ F \circ T_2$$

Le filtre qu'ils proposent est présenté en Figure 3.2. Dans ce filtre, on constate que les transitions ϵ sont différenciées en ϵ_1 et ϵ_2, où ϵ_1 identifie un ϵ qui étiquette l'entrée d'une transition, et ϵ_2 identifie un ϵ qui étiquette la sortie d'une transition. A partir de l'état 0 du filtre, toutes les transitions qui combinent des symboles ϵ sont possibles. On ne quitte l'état 0 pour aller en 1 qu'avec une transition ϵ_1:ϵ_1. Lorsque l'on se trouve à l'état 1, on constate que seuls les symboles de l'alphabet (Σ) permettent de revenir à l'état 0, et que toute transition étiquetée en entrée ou en sortie par ϵ_2 est impossible. Une analyse similaire peut être réalisée pour l'état 2, qui interdit toute transition étiquetée en entrée ou en sortie par ϵ_1. L'objectif du filtre est donc d'*interdire toute suite de transitions dans lesquelles le symbole ϵ alterne entre l'entrée et la sortie.*

Le filtre n'est cependant utilisable que si les transitions des transducteurs ont été modifiées. La Figure 3.3 montre le résultat de la modification des transducteurs, et la composition qui en résulte. Dans T_1', les transitions ϵ en sortie sont modifiées en ϵ_2, et des cycles $\epsilon : \epsilon_1$ sont ajoutés à tous les états. Dans T_2', les transitions ϵ en entrée sont modifiées en ϵ_1, et des cycles $\epsilon_2 : \epsilon$ sont ajoutés à tous les états. Grâce à ces modifications, et à la composition des transducteurs modifiés au travers du filtre, un seul chemin est autorisé.

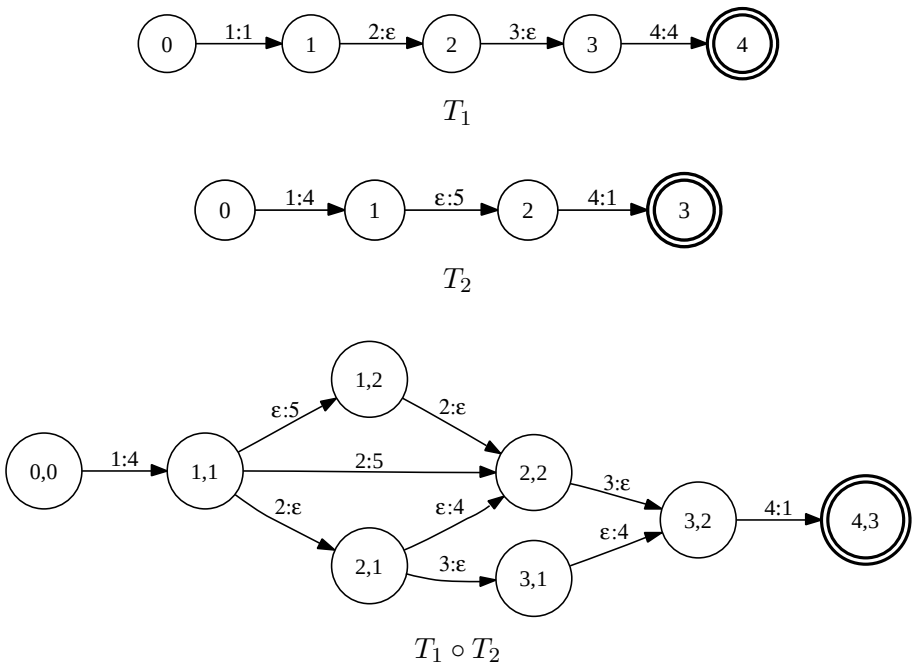


FIG. 3.1: Composition naïve avec transitions ϵ. Le graphe $T_1 \circ T_2$ autorise de nombreux chemins entre les états $(1, 1)$ et $(3, 2)$

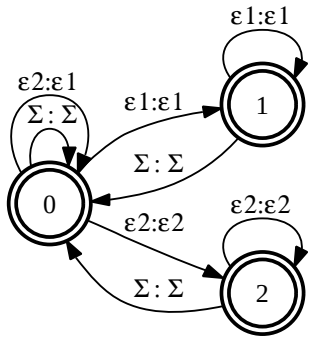


FIG. 3.2: Filtre de composition. Bien que l'idée d'un filtre de composition vienne de Pereira & Riley (1997), le filtre présenté ici en est une amélioration proposée par Mohri *et al.* (1996)

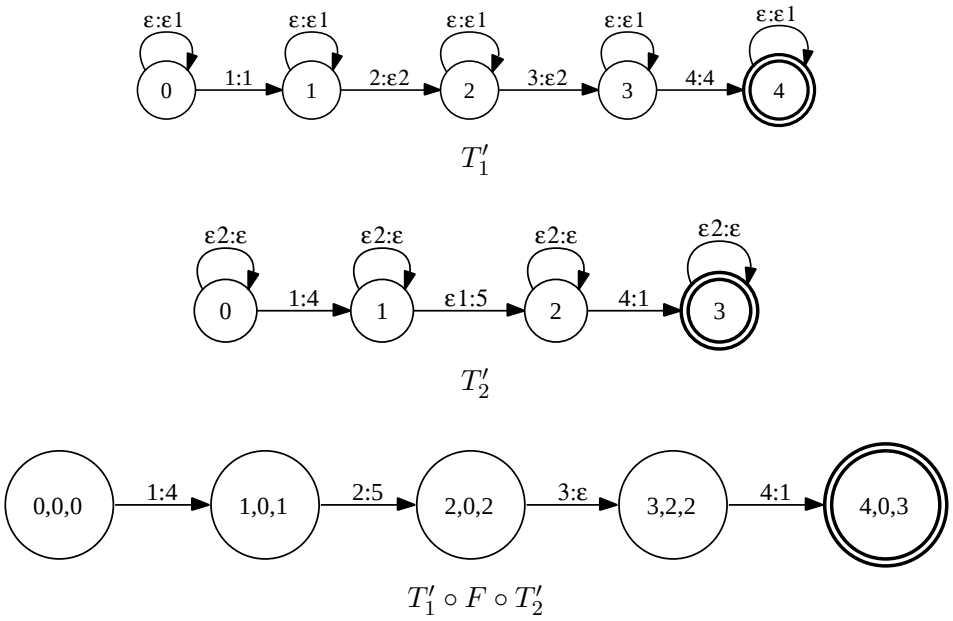


FIG. 3.3: Composition filtrée avec transitions ϵ

Propriétés de la composition. La composition est associative : en effet, dans une cascade de compositions, l'ordre d'association de l'opérateur $\circ$ importe peu. Par exemple,

$$\begin{aligned} T_1 \circ T_2 \circ T_3 \circ T_4 &= (T_1 \circ T_2) \circ (T_3 \circ T_4) \\ &= T_1 \circ (T_2 \circ T_3) \circ T_4 \\ &= (T_1 \circ T_2 \circ T_3) \circ T_4 \\ &= T_1 \circ (T_2 \circ T_3 \circ T_4) \end{aligned}$$

La composition n'est par contre pas commutative : étant donné deux alphabets différents Σ_1 et Σ_2, et deux transducteurs $T_1 = (\Sigma_1, \Sigma_2, Q_1, i_1, F_1, E_1)$ et $T_2 = (\Sigma_2, \Sigma_1, Q_2, i_2, F_2, E_2)$, $T_1 \circ T_2 \neq T_2 \circ T_1$.

Transitions vides. Notons que même si les transducteurs qui sont composés ne présentent pas de transitions vides (*i.e.* étiquetées par la paire (ϵ, ϵ)), le transducteur résultant peut en contenir :

$$\forall a \in \Sigma,\ (p_1, \epsilon, a, q_1) \circ (p_2, a, \epsilon, q_2) = ((p_1, p_2), \epsilon, \epsilon, (q_1, q_2))$$

Il sera dès lors utile de les supprimer (cf. Section 2.3.3) afin d'éviter tout délai lors du parcours d'une string d'entrée.

3.3.4 Clôture sous l'intersection

Etant donné deux transducteurs lettre-à-lettre sans ϵ, T_1 et T_2, il est possible de calculer un transducteur $T_1 \cap T_2$ tel que $|T_1 \cap T_2| = |T_1| \cap |T_2|$. L'intersection de deux transducteurs lettre-à-lettre sans ϵ s'obtient en calculant l'intersection de leurs automates sous-jacents.

Un transducteur avec ϵ n'assure pas la clôture sous l'intersection. La Figure 3.4 illustre ce problème. T_1 modélise la transduction $|T_1|(c^n) = \{A^n B^m \mid m \geq 0\}$, tandis que T_2

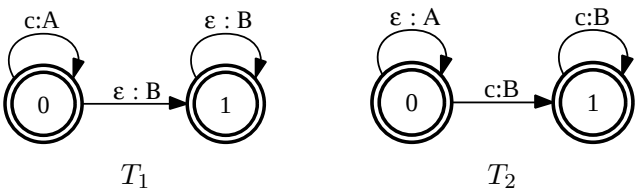


FIG. 3.4: Transducteurs à l'intersection non régulière

modélise la transduction $|T_2|(c^n) = \{A^m B^n \mid m \geq 0\}$. L'intersection de ces transducteurs doit être telle que $|T_1|(c^n) \cap |T_2|(c^n) = |T_1 \cap T_2|(c^n)$, c'est-à-dire $\{A^n B^n \mid n \geq 0\}$. Si elle est calculable, cette intersection n'est cependant pas représentable sous la forme d'un transducteur, étant donné que $A^n B^n$ n'est pas un langage régulier, mais un langage non-contextuel qui nécessite au moins un automate à pile (cf. Section 1.3.2).

3.4 Transducteurs séquentiels

3.4.1 Définition

Définition 3.4.1. *Un* transducteur séquentiel T *est un 7-uplet* $(\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma)$ *où* Σ_1, Σ_2, Q, i *et* F *sont définis comme pour un transducteur,* δ *est la fonction de transition qui projette* $Q \times \Sigma_1$ *sur* Q *et* σ *est la fonction d'émission qui projette* $Q \times \Sigma_1$ *sur* Σ_2^*.

Comme l'illustre la Figure 3.5, un transducteur séquentiel est un transducteur dont *l'entrée est déterministe*. Dans un transducteur de ce type, un symbole de l'alphabet d'entrée étiquette au plus un arc d'un état donné. La sortie d'un transducteur séquentiel n'est par

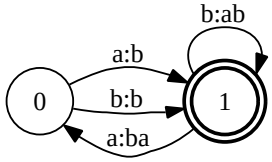


FIG. 3.5: Transducteur séquentiel T. Figure reprise de (Mohri 1997b)

contre pas forcément déterministe, et les arcs peuvent être étiquetés en sortie par des strings.

Les transducteurs séquentiels sont très intéressants, parce qu'ils assurent un temps de calcul linéairement proportionnel à la taille de l'entrée à analyser.

Comme dans le cas des transducteurs, les fonctions de transition étendue $\hat{\delta}$ et d'émission étendue $\hat{\sigma}$ peuvent être définies de sorte que :

(i) $\forall q \in Q$,

- $\hat{\delta}(q, \epsilon) = q$
- $\hat{\sigma}(q, \epsilon) = \epsilon$

(ii) $\forall u_1 \in \Sigma_1^*$ et $\forall a_1 \in \Sigma_1$,

- $\hat{\delta}(p, u_1 \cdot a_1) = \delta(\hat{\delta}(p, u_1), a_1)$
- $\hat{\sigma}(p, u_1 \cdot a_1) = \hat{\sigma}(p, u_1) \cdot \sigma(\hat{\delta}(p, u_1), a_1)$

Schützenberger (1977) a démontré que les transducteurs séquentiels peuvent être généralisés en ajoutant la possibilité qu'un état final produise une string de sortie supplémentaire. On parle de transducteur *sous-séquentiel*.

Définition 3.4.2. *Un* transducteur sous-séquentiel T *est un 8-uplet* $(\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma, \varphi)$ *où* $(\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma)$ *est un transducteur séquentiel, et* φ *est la fonction d'émission finale qui projette* F *sur* Σ_2^*.

Le résultat de la composition d'une string u avec un transducteur sous-séquentiel T, si le parcours de la string se termine à un état final, est la string $\hat{\sigma}(u) \cdot \varphi(\hat{\delta}(u))$. Etant donné que φ projette F sur Σ_2^*, $\varphi(\hat{\delta}(u))$ peut valoir ϵ.

La Figure 3.6 donne un exemple de transducteur sous-séquentiel. Le parcours de la string *ab* donnera en sortie *bab*, tandis que le parcours de la string *abc* donnera en sortie *baca*. On constate que la sous-séquentialisation permet à l'entrée de rester déterministe, malgré les divergences des strings de sortie.

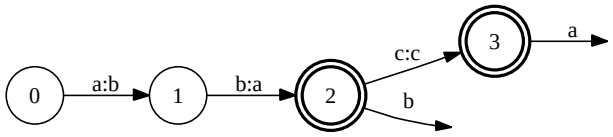


FIG. 3.6: Transducteur sous-séquentiel T_1

Cependant, l'ambiguïté du langage naturel ne se satisfait pas des possibilités offertes par les transducteurs séquentiels. Il arrive en effet souvent qu'un niveau d'analyse donné propose p analyses pour une entrée donnée. Fort de ce constat, Mohri (1994a) a proposé une extension des transducteurs sous-séquentiels, autorisant l'ajout de p strings de sortie supplémentaires à chaque état final. On parle de transducteur *p-sous-séquentiel*.

Définition 3.4.3. *Un transducteur p-sous-séquentiel T est un 8-uplet* $(\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma, \Phi)$ *où* $(\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma)$ *est un transducteur séquentiel, et* Φ *est la fonction d'émission finale qui projettent* F *sur* $(\Sigma_2^*)^p$.

Le résultat de la composition d'une string u avec un transducteur p-sous-séquentiel T, si le parcours de la string se termine à un état final, est un ensemble de maximum p strings distinctes $\{\hat{\sigma}(u) \cdot \Phi_i(\hat{\delta}(u)) \mid \hat{\delta}(u) \in F \text{ et } 1 \leq i \leq p\}$.

La Figure 3.7 donne un exemple de transducteur p-sous-séquentiel. Ce transducteur, pour lequel $p = 2$, présente les différentes analyses possibles des sous-strings de « *couvent* » en terme d'attribution des catégories `NOUN/VERB`. On constate que l'entrée de la machine reste déterministe.

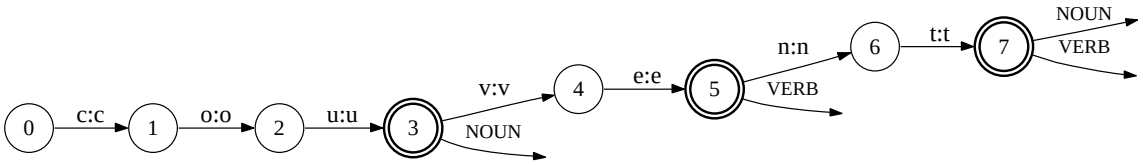


FIG. 3.7: Transducteur p-sous-séquentiel T_2, avec $p = 2$

Note 3.4.1. Dans la suite de ce document, nous parlons de *transducteur de type séquentiel* lorsque nous faisons référence, sans distinction, à un transducteur dont l'entrée est déterministe, qu'il soit séquentiel *stricto sensu*, sous-séquentiel ou p-sous-séquentiel.

3.4.2 Séquentialisation d'un transducteur

Nous avons mentionné en Section 3.2 que la déterminisation d'un transducteur au travers de son automate sous-jacent, contrairement à la déterminisation d'un automate, n'assure pas pour autant un temps de calcul linéairement proportionnel à la taille de l'entrée à analyser. Ceci est dû au fait que, étant donné $T = (\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma)$, la déterminisation de T au travers de son automate sous-jacent assure uniquement que δ projette $Q \times (\Sigma_1 \times \Sigma_2)$ sur Q. Par contre, δ projette toujours potentiellement $Q \times \Sigma_1$ sur $\mathcal{P}^Q$. En d'autres termes, la déterminisation d'un transducteur au travers de son automate sous-jacent n'assure pas que le résultat soit de type séquentiel.

Pour cette raison, Mohri (1996) a proposé un algorithme de séquentialisation d'un transducteur.

Note 3.4.2. Nous parlons de *séquentialisation* alors que Mohri parle de *déterminisation d'un transducteur*. Nous préférons le terme *séquentialisation* parce qu'il indique sans ambiguïté qu'il s'agit d'obtenir un transducteur de type séquentiel.

L'algorithme de Mohri assure l'obtention d'un transducteur de type séquentiel à partir d'un transducteur classique, pour autant que celui-ci soit séquentialisable. En effet,

tous les transducteurs ne sont pas séquentialisables. Ceci est dû au fait que les transducteurs de type séquentiel constituent une sous-classe stricte de tous les transducteurs, comme l'a démontré Choffrut (1978). Notons que tout transducteur acyclique est séquentialisable, résultat particulièrement intéressant pour le traitement du langage naturel, où les données à analyser sont par essence acycliques.

L'algorithme de Mohri repose sur les notions de *préfixe* et d'*inverse* d'une string.

Définition 3.4.4. *Etant donné deux strings x et y, x est un* préfixe *de y pour autant qu'il existe une string u tel que $y = xu$, de sorte que $x \leq y$. Le préfixe de deux strings x et y se note $x \wedge y$. Le préfixe d'un ensemble de strings X se note $\bigwedge X$.*

Définition 3.4.5. *Etant donné deux strings x et y, $x^{-1}y = u$ est l'*inverse *de y pour autant qu'il existe une string u tel que $y = xu$. Par exemple, si $y = abcd$ et $x = ab$, $(ab)^{-1}abcd = cd$.*

Etant donné un transducteur $T = (\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma)$, l'algorithme va construire un transducteur $T' = (\Sigma_1, \Sigma_2, Q', i', F', \delta', \sigma', \Phi)$. Cet algorithme s'inspire de l'algorithme de déterminisation des automates (cf. Pseudocode 1), établi sur la construction de $\mathcal{P}^Q$, l'ensemble des parties de Q. La différence est qu'ici chaque état p d'un ensemble $p' \in \mathcal{P}^Q$ est associé à une string $u : p' = \{(p, u)\}$. Cette string correspond à un *retard* dans l'émission des sorties du transducteur, retard dû au fait que les sorties correspondant à un couple (état, symbole) donné peuvent être différentes. A cause de cette différence, seul le plus long préfixe commun de ces strings de sortie est gardé sur la transition, tandis que l'inverse de chaque string est reporté sur l'état atteint, ce qui explique que les sous-ensembles de $\mathcal{P}^Q$ ne contiennent plus des états simples, mais des couples (état, string).

Le Pseudocode 13 présente l'algorithme de séquentialisation. Le premier état p' construit est l'état initial de T', formé du couple (i, ϵ) (ligne 2). La string associée à cet état est vide puisqu'aucun retard n'a encore été observé. A chaque boucle, un nouvel état p' est traité (ligne 4). L'état p' est final s'il contient au moins un couple (p, u) dont p est final dans T (lignes 8–9). Dans ce cas, pour tout couple final (p, u), u est ajouté à l'ensemble des sorties finales de p' (ligne 10). Ensuite, chaque symbole a_1 de l'alphabet d'entrée est traité (lignes 12–33), pour autant qu'au moins une transition étiquetée a_1 quitte au moins un état p de p'. La tâche de l'algorithme est de construire $\delta'(p', a_1)$ et $\sigma'(p', a_1)$. Pour ce faire, la première étape est de définir le préfixe commun aux différentes transitions étiquetées a_1 quittant les états de p' (lignes 15–23). Ce préfixe commun sera attribué ultérieurement à $\sigma'(p', a_1)$ (ligne 31). La seconde étape est de construire l'état q' (lignes 24–26). Pour chaque état q atteint depuis un état p de p' à l'aide de a_1, on construit une string u_2 qui correspond à l'inverse u'^{-1} $(u \cdot \sigma(p, a_1))$ (ligne 25). L'état $q' = \{(q, u_2)\}$ est ajouté aux états de Q' pour autant qu'il soit nouveau (lignes 28–29), et est attribué à $\delta'(p', a_1)$ (ligne 30).

La Figure 3.8 illustre le déroulement de l'algorithme sur un état. Dans le graphe de gauche, on constate que $\sigma(p, a) = \{ab, ac, ad\}$. Le plus long préfixe commun de ces sorties est a, qui sera gardé sur la transition. Les inverses de ces strings, respectivement b, c et d, sont associées aux états atteints dans le sous-ensemble de $\mathcal{P}^Q$ constitué : $\{(q, b), (r, c), (r, d)\}$. En

Require: Un FST $T = (\Sigma_1, \Sigma_2, Q, i, F, \delta, \sigma)$
Ensure: Le FST de type séquentiel $T' = (\Sigma_1, \Sigma_2, Q', i', F', \delta', \sigma', \Phi)$ équivalent à T

1: $H \leftarrow F' \leftarrow E' \leftarrow \emptyset$
2: $p' \leftarrow i' \leftarrow$ `INSERT`$(H, \{(i, \epsilon)\})$ /* 1er élément $\rightarrow$ retour $= 0$ */
3: $Q' \leftarrow \{i'\}$
4: **while** $p' < |Q'|$ **do**
5: $S_{p'} \leftarrow$ `GET`(H, p')
6: $\Phi(p') \leftarrow \emptyset$
7: $F_{p'} \leftarrow \{(p, u) \in S_{p'} : p \in F\}$
8: **if** $F_{p'} \neq \emptyset$ **then**
9: $F' \leftarrow F' \cup \{p'\}$
10: $\Phi(p') \leftarrow \Phi(p') \cup \{u\}$ **for each** $(p, u) \in F_{p'}$
11: **end if**
12: **for each** $a_1 \in \Sigma_1$ **do**
13: $S_{q'} \leftarrow \emptyset$
14: $u' \leftarrow$ `UNDEF`
15: **for each** $(p, u) \in S_{p'}$ **do**
16: **for each** $a_2 \in \sigma(p, a_1)$ **do**
17: **if** $u' =$ `UNDEF` **then**
18: $u' \leftarrow u \cdot a_2$
19: **else**
20: $u' \leftarrow u' \wedge u \cdot a_2$
21: **end if**
22: **end for**
23: **end for**
24: **for each** $(p, u) \in S_{p'}$ **do**
25: $S_{q'} \leftarrow S_{q'} \cup \{(q, u'^{-1}u \cdot \sigma(p, a_1))\}$ **for each** $q \in \delta(p, a_1)$
26: **end for**
27: **if** $S_{q'} \neq \emptyset$ **then**
28: $q' \leftarrow$ `INSERT`$(H, S_{q'})$
29: $Q' \leftarrow Q' \cup \{q'\}$
30: $\delta'(p', a_1) \leftarrow q'$
31: $\sigma'(p', a_1) \leftarrow u'$
32: **end if**
33: **end for**
34: $p' = p' + 1$
35: **end while**
36: $T' \leftarrow (\Sigma_1, \Sigma_2, Q', i', F', \delta', \sigma', \Phi)$
37: **return** T'

Pseudocode 13: `SEQUENTIALISATION`

outre, les états r et s sont finaux. De ce fait, le sous-ensemble est final, et les strings associées à r et s, respectivement c et d, deviennent les strings finales émises par le sous-ensemble. Le graphe de droite montre le résultat de l'exécution de l'algorithme sur l'état p.

3.4.3 Minimisation d'un transducteur séquentiel

3.4.3.1 Introduction

Nous avons mentionné en Section 3.2 que la minimisation d'un transducteur est réalisable au travers de son automate sous-jacent. Cependant, Choffrut (1978) a démontré qu'un transducteur de type séquentiel peut être minimisé de manière plus efficace, pour autant que l'on ajoute une étape préliminaire à la minimisation classique : cette étape consiste à construire le *préfixe de l'automate de sortie* du transducteur. Pour tout sous-ensemble L de Σ^*, et toute string u, u^{-1} note le préfixe de L tel que :

$$u^{-1}L = \{v : uv \in L\} \tag{3.4.3.1}$$

Choffrut fonde sa démonstration sur les résultats de Nérode (1958), qui démontre que si L est un langage régulier, alors il existe k sous-ensembles $u^{-1}L$ distincts. Dans l'automate minimal équivalent à L, chaque état correspondra à un ensemble $u^{-1}L$ donné.

Note 3.4.3. On pose un automate $A = (\Sigma, Q, i, F, E)$ dont tous les états sont coaccessibles : chaque état admet au moins un chemin vers un état final. Pour tout état $p \in Q$, P est la fonction projetant Q sur Σ^* qui associe à tout état p *le plus long préfixe commun* de tous les chemins allant de p à un état final. La définition récursive suivante est plus précise :

$$\forall p \in (Q - F),\ P(p) = \bigwedge_{e \in E[p]} e.a \cdot P(e.q), \tag{3.4.3.2}$$
$$\forall p \in (F),\ P(p) = \epsilon.$$

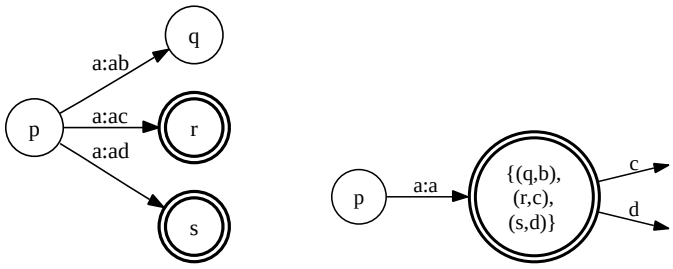
On constate que P est correctement définie puisque tous les états admettent au moins un chemin rejoignant un état final.

Définition 3.4.6. *Etant donné un automate $A = (\Sigma, Q, i, F, E)$, le* préfixe *de A est l'automate $A' = (\Sigma, Q, i, F, E')$ tel que :*

$$E' = \{(p, P(p)^{-1}x \cdot P(q), q) \mid (p, x, q) \in E\}$$

On peut vérifier que si (p, x, q) est une transition de E, alors $P(p)$ est par définition un préfixe de $x \cdot P(q)$, ce qui assure la consistance de la définition précédente.

En somme, A' possède le même graphe que A et reconnaît le même langage, mais est étiqueté différemment.



état p déterministe *état p séquentiel*

FIG. 3.8: Séquentialisation d'un état

Le premier algorithme calculant le préfixe d'un automate est celui de Mohri (1994a, 2000), appelé *quasi-déterminisation*. Mohri a remarqué que cette première étape dans la minimisation d'un transducteur de type séquentiel est indépendante de la notion de transducteur. La quasi-déterminisation est un algorithme qui travaille sur des automates. Il garde le graphe de l'automate dont il change uniquement les étiquettes des arcs. Informellement, il pousse autant que possible les étiquettes des arcs des états finaux vers l'état initial. Un autre algorithme de calcul du préfixe d'un automate a été proposé par Breslauer (1995). L'algorithme se base sur la construction de l'arbre-suffixe d'un arbre. La complexité de cet algorithme est cependant dépendante de la taille de l'alphabet de l'automate. La comparaison de la complexité des deux algorithmes (Mohri 2000) montre que Breslauer est meilleur uniquement lorsque le nombre de transitions de la machine est réduit. Dans tous les autres cas, Mohri a l'avantage.

Béal & Carton (2000) font cependant remarquer qu'il existe un cas où l'algorithme de Mohri ne fonctionne pas : lorsque l'automate contient un cycle étiqueté ϵ, l'algorithme n'admet pas une seule solution. L'algorithme n'est donc valable que lorsque l'automate ne présente pas de cycle étiqueté ϵ, ce qui n'est pas restrictif pour les applications dans le domaine du langage naturel, qui ne présentent jamais de cycles ϵ en sortie. Béal propose cependant un algorithme de même complexité que celui de Mohri, valable pour tous les automates.

Or, l'algorithme de Mohri présente de fortes similitudes avec d'autres algorithmes qu'il a proposés dans le contexte des machines pondérées, algorithmes qui n'ont pas d'équivalents (cf. Section 4.6). En outre, nos applications sont centrées sur le langage naturel, et ne sont donc pas concernées par les limites mises en évidence par Béal. Nous décrivons donc l'algorithme de Mohri, étant donné qu'il nous a permis d'homogénéiser les algorithmes de notre propre bibliothèque (cf. Chapitre 6).

3.4.3.2 Principe de l'algorithme

La définition de la fonction P suggère qu'avant de calculer $P(p)$, il faut calculer P pour tous les états de la liste d'adjacence de P.

Définition 3.4.7. *Une* liste d'adjacence *est une structure de données qui permet de représenter les connexions entre les états d'un graphe en mémorisant pour chaque état du graphe une liste des états vers lesquels il a une transition.*

Ceci nécessite un classement linéaire de tous les états de A, ce qui n'est possible que si A ne contient pas de cycle.

Si A est acyclique, un classement linéaire des états existe, et correspond à l'ordre inverse d'un tri topologique du graphe orienté, obtenu en un temps linéaire $O(|Q| + |E|)$, en établissant l'ordre inverse d'une recherche en profondeur d'abord du graphe (Aho *et al.* 1974).

Définition 3.4.8. *Un* tri topologique *d'un graphe orienté est un classement linéaire de ses états, compatible avec l'ordre partiel R induit sur les états : p vient avant q (pRq) s'il existe une transition de p vers q dans le graphe orienté.*

En respectant cet ordre, on peut calculer le préfixe de chaque état de A au plus une fois, et construire l'automate-préfixe de A.

Si A n'est pas acyclique, il est nécessaire de considérer ses composants fortement connectés.

Définition 3.4.9. *Etant donné un graphe orienté $A = (Q, E)$, il est possible de partitionner Q en classes d'équivalence Q_i, $1 \leq i \leq n$, tel que les états p et q sont équivalents si et seulement si pRq et qRp. Posons E_i, $1 \leq i \leq n$, l'ensemble de transitions connectant les paires d'états de Q_i. Les graphes (Q_i, E_i) sont appelés les* composants fortement connectés *(SCC [1]) de A. Même si tous les états de A sont dans un Q_i donné, des transitions de A peuvent n'être dans aucun E_i. Un graphe est dit* fortement connecté *s'il possède un seul composant fortement connecté.*

Un graphe orienté peut être décomposé en ses composants fortement connectés (Aho *et al.* 1974) sous la forme d'un graphe des composants de A, que nous notons $A^{SCC} = (Q^{SCC}, E^{SCC})$, où Q^{SCC} est l'ensemble des composants fortement connectés $\{scc_1, \ldots, scc_n\}$ et E^{SCC} est l'ensemble des transitions. Etant donné que le graphe des composants est un graphe orienté acyclique, il existe un classement linéaire des composants fortement connectés tel que si la liste d'adjacence d'un état $p \in scc_1$ contient un état $q \in scc_2$, alors scc_2 apparaît avant scc_1 dans le classement. Ce classement est l'ordre inverse d'un tri topologique de A^{SCC}. En respectant cet ordre, on peut alors calculer le préfixe de chaque état de A^{SCC} au plus une fois, et construire l'automate-préfixe de A^{SCC}. Nous procédons donc comme dans le cas d'un graphe acyclique. Par contre, il est dans ce cas-ci également nécessaire de modifier les transitions de chaque SCC.

[1] *SCC* pour *Strongly Connected Component* en anglais.

Note 3.4.4. Le raisonnement qui suit contient les notations que voici :

- $\forall p \in Q$, $E^R[p]$ note l'ensemble des transitions $e \in E$ qui atteignent p.
- $\forall p \in scc_i$,

$$In(p) = \{e \in E[p] \mid e.q \in scc_i\},$$
$$Out(p) = \{e \in E[p] \mid e.q \notin scc_i\}.$$

Etant donné un SCC scc_i dont tous les prédécesseurs ont été modifiés correctement [2], le but est de calculer le plus long préfixe commun de chaque état p appartenant à scc_i : $P(p) = X_p$, qui est résolu selon le système d'équations suivant :

$$\begin{aligned}
&si \quad p \in F, \quad X_p = \epsilon,\\
&sinon, \quad si \quad Out(p) = \emptyset, \quad X_p = \left(\bigwedge_{e \in E[p]} e.a \cdot X_{e.q}\right)\\
&sinon, \quad X_p = \left(\bigwedge_{e \in In(p)} e.a \cdot X_{e.q}\right) \wedge \left(\bigwedge_{e \in Out(p)} e.a\right)
\end{aligned} \tag{3.4.3.3}$$

La résolution de X_p pour chaque état $p \in scc_i$ modifie de la façon souhaitée l'ensemble des transitions de scc_i ainsi que les transitions entrant dans scc_i. Mohri démontre que cette modification peut être réalisée de manière itérative, en faisant évoluer les valeurs de quelques variables.

Base. Pour chaque état $p \in scc_i$, on définit π_p tel que

$$\begin{aligned}
&\pi_p = \left(\bigwedge_{e \in E[p]} e.a\right) \quad si \quad p \notin F,\\
&\pi_p = \epsilon \quad sinon.
\end{aligned} \tag{3.4.3.4}$$

Etant donné un état $p_0 \in scc_i$ tel que $\pi_{p_0} \neq \epsilon$, on peut considérer que p_0 est un *candidat pour un changement de variable :*

$$X_{p_0} \leftarrow \pi_{p_0} Y_{p_0}$$

où $(Y_p)_{p \in scc_i}$ est l'ensemble des variables inconnues. On peut dès lors modifier l'ensemble des transitions comme suit :

$$\begin{aligned}
&\forall e \notin (E[p_0] \cup E^R[p_0]), \quad e.a' = e.a,\\
&\forall e \in E[p_0] - (E[p_0] \cap E^R[p_0]), \quad e.a' = \pi_{p_0}^{-1}\, e.a,\\
&\forall e \in E^R[p_0] - (E[p_0] \cap E^R[p_0]), \quad e.a' = e.a \cdot \pi_{p_0},\\
&\forall e \in (E[p_0] \cap E^R[p_0]), \quad e.a' = \pi_{p_0}^{-1}\, e.a \cdot \pi_{p_0}.
\end{aligned} \tag{3.4.3.5}$$

[2] Ceci est nécessairement vrai pour le premier SCC de la liste, puisqu'aucune transition n'en sort.

Ce qui peut se simplifier :

$$\begin{array}{l}\forall e \in E,\\ \quad si \quad e \notin (E[p_0] \cup E^R[p_0]), \quad e.a' = e.a,\\ \quad sinon \begin{cases} si \quad e \in E[p_0], \quad e.a' = \pi_{p_0}^{-1}\, e.a,\\ si \quad e \in E^R[p_0], \quad e.a' = e.a \cdot \pi_{p_0}. \end{cases}\end{array} \tag{3.4.3.6}$$

Dans ce changement de variable, Y_p est le nouveau système :

$$\begin{array}{l} si \quad p \in F, \quad Y_p = \epsilon,\\ sinon, \quad si \quad Out(p) = \emptyset, \quad Y_p = \left(\bigwedge_{e \in E[p]} e.a' \cdot Y_{e.q} \right)\\ sinon, \quad Y_p = \left(\bigwedge_{e \in In(p)} e.a' \cdot Y_{e.q} \right) \wedge \left(\bigwedge_{e \in Out(p)} e.a' \right) \end{array} \tag{3.4.3.7}$$

Induction. Pour autant que l'on fasse un nouveau changement de variable basé (en 3.4.3.6) sur $e.a'$ au lieu de $e.a$, le système 3.4.3.7 peut devenir itératif et être répété tant qu'il existe un état $p \in scc_i$ tel que $\pi_p \neq \epsilon$.

Au travers de ces itérations successives, les changements successifs de variable modifient exactement les transitions de chaque SCC de l'automate A en celles du préfixe A', ce qui est le but poursuivi.

3.4.3.3 Algorithme

L'algorithme de préfixation est présenté en Pseudocode 14. Etant donné un automate $A = (\Sigma, Q, i, F, E)$ et un ensemble E' initialisé comme une copie de E (ligne 2), l'objectif de l'algorithme est de modifier les transitions de E' de manière à ce qu'il représente les transitions de l'automate préfixe de A. L'algorithme itère sur chaque SCC s (ligne 7) contenu dans le vecteur SCC_A, classé dans l'ordre inverse du tri topologique des états de A (ligne 1).

Pour chaque SCC s, l'algorithme initialise le changement de variable à partir d'un état p de s, choisi au hasard et placé dans une file C (ligne 8) dont nous expliquons l'intérêt ci-dessous.

Le changement de variable réalise en deux étapes les modifications de transitions présentées en 3.4.3.6. L'algorithme commence par calculer π pour p (ligne 12) *via* la procédure `LCP` [3] présentée en Pseudocode 15. Si $\pi \neq 0$, la procédure `LCP` supprime π à la gauche des transitions qui quittent p (`LCP`, lignes 12–17). De retour de la procédure et si $\pi \neq 0$, l'algorithme concatène π à la droite des transitions qui atteignent p (lignes 13–23).

[3] Abréviation de l'anglais *Longest Common Prefix*, « plus long préfixe commun ».

Require: Un FSA $A = (\Sigma, Q, i, F, E)$
Ensure: Le FSA $A' = (\Sigma, Q, i, F, E')$, automate-préfixe de A
1: $SCC_A \leftarrow$ `INVERSE`(`TOPOLOGICAL_SORT`(`SCC`(A)))
2: $E' \leftarrow$ `COPY`(E)
3: $A' \leftarrow (\Sigma, Q, i, F, E')$
4: $V_\epsilon[p] \leftarrow 0$ **for** $p \leftarrow 0$ **to** $|Q| - 1$
5: $V_f[p] \leftarrow 0$ **for** $p \leftarrow 0$ **to** $|Q| - 1$
6: $C \leftarrow \emptyset$
7: **for** $s \leftarrow 0$ to $|SCC_A| - 1$ **do**
8: `ENQUEUE`(C, `GET`($SCC_A[s]$))
9: **while** $C \neq \emptyset$ **do**
10: $p \leftarrow$ `DEQUEUE`(C)
11: **if** $E'^R[p] \neq \emptyset$ **then**
12: $\pi \leftarrow$ `LCP`(A', p, V_ϵ, V_f)
13: **for each** $e \in E'^R[p]$ **do**
14: **if** $\pi \neq \epsilon$ **then**
15: **if** $e.q \in SCC_A[s]$ **and** $e.a = \epsilon$ **and** $V_\epsilon[e.q] > 0$ **and** $V_f[e.q] = 0$ **then**
16: $V_\epsilon[e.q] \leftarrow V_\epsilon[e.q] - 1$
17: **end if**
18: $e.a \leftarrow e.a \cdot \pi$
19: **end if**
20: **if** `INQUEUE`($C, e.q$) = `FALSE` **and** $V_\epsilon[e.q] = 0$ **and** $V_f[e.q] = 0$ **then**
21: `ENQUEUE`($C, e.q$)
22: **end if**
23: **end for**
24: **end if**
25: **end while**
26: **end for**
27: **return** A'

Pseudocode 14: `PREFIXATION`

La procédure `LCP` est évidemment coûteuse. L'idée est donc d'y recourir aussi peu que possible. Afin d'éviter le traitement d'états qui ne peuvent être candidats à un changement de variable, le principe est de ne placer dans une file C que les états candidats à un changement de variable. Pour initialiser le processus, un état p choisi au hasard dans le SCC s est placé dans la file. Ensuite, on n'ajoute que les états q dont au moins une transition, atteignant l'état p en cours de traitement, vient d'être modifiée (ligne 21). L'algorithme s'interrompt donc naturellement lorsque la file est vide (ligne 9). Cependant, certains de ces états ne peuvent plus être candidats, soit parce qu'ils présentent des transitions ϵ, soit parce qu'ils sont finaux. Deux vecteurs permettent de mémoriser ces informations :

- V_ϵ, dans lequel la procédure `LCP` mémorise pour chaque état le nombre de transitions devenues ϵ après suppression à gauche de π (`LCP`, ligne 15),
- V_f, dont $V_f[p]$ vaut 1 si p est final ou si $V_f[p]$ vaut toujours 0 après le calcul de π.

Les états ne sont donc ajoutés dans la file que s'ils ne présentent pas de transition ϵ et ne sont pas finaux (ligne 20).

3.5 Synthèse

Les transducteurs à états finis, vus comme des graphes orientés étiquetés, sont simplement des automates dont les transitions ont été augmentées de symboles de sortie. Un automate peut d'ailleurs être transformé en transducteur identitaire, et les première et seconde projections d'un transducteur sont des automates. En outre, tous les algorithmes définis sur les automates sont applicables au transducteur, pour autant qu'il soit considéré *via* son automate sous-jacent.

Cependant, la déterminisation d'un transducteur n'assure pas un parcours linéairement proportionnel à la longueur de la string d'entrée. Or, il existe un type particulier de transducteur : le transducteur de type séquentiel, dont l'entrée déterministe assure un parcours linéairement proportionnel à la taille de l'entrée à analyser, comme dans le cas des automates déterministes. Dans un souci d'efficacité, des algorithmes ont été définis afin de rendre séquentiel un transducteur, et de minimiser de manière optimale un transducteur de type séquentiel.

Les transducteurs peuvent également être vus comme définissant des relations et des transductions entre langages réguliers. Dans ce contexte, une opération régulière majeure s'applique aux transducteurs : la composition, généralisation de l'intersection des automates, qui permet de combiner différents niveaux de représentation pour construire des relations complexes à partir de relations simples.

En traitement du langage naturel, la composition représente donc une solution aisée afin de combiner l'information répartie entre divers niveaux linguistiques. De ce fait, les transducteurs ont été et sont encore énormément employés dans de nombreux domaines du traitement automatique des langues, comme l'analyse lexicale (Silberztein 1993, Karttunen

1994), l'analyse morphologique et phonologique (Karttunen *et al.* 1992, Kaplan & Kay 1994, Mohri & Sproat 1996), ou l'analyse syntaxique (Roche 1993, Mohri 1994b).

Require: Un FSA $A = (\Sigma, Q, i, F, E)$, un état $p \in Q$, V_ϵ, V_f
Ensure: π le plus long préfixe commun des transitions quittant p

1: **if** $p \in F$ **then**
2: $\pi \leftarrow \epsilon$
3: $V_f[p] \leftarrow 1$
4: **else**
5: $\pi \leftarrow E[p][0]$
6: $t \leftarrow 1$
7: **while** $t < |E[p]| - 1$ **and** $\pi \neq \epsilon$ **do**
8: $\pi \leftarrow \pi \wedge E[p][t].a$
9: $t \leftarrow t + 1$
10: **end while**
11: **if** $\pi \neq \epsilon$ **then**
12: **for each** $e \in E[p]$ **do**
13: $e.a \leftarrow \pi^{-1} e.a$
14: **if** $e.a = \epsilon$ **then**
15: $V_\epsilon[p] \leftarrow V_\epsilon[p] + 1$
16: **end if**
17: **end for**
18: **end if**
19: **if** $V_\epsilon[p] = 0$ **then**
20: $V_f[p] \leftarrow 1$
21: **end if**
22: **end if**
23: **return** π

Pseudocode 15: `LCP`

Chapitre 4

Les machines pondérées

4.1 Introduction

La puissance des automates et des transducteurs ne suffit pas à certains domaines du traitement automatique des langues, qui ont besoin d'outils capables de gérer ce que Mohri *et al.* (2000) appellent *un certain degré d'incertitude*. C'est le cas, par exemple, de la reconnaissance de la parole. C'est le cas, comme nous le montrons dans la suite de ce document, de la synthèse par sélection d'unités non uniformes (Partie II) et de la correction orthographique (Partie III).

Certains auteurs (Pereira & Sproat 1994, Pereira & Riley 1997) décrivent l'incertitude dont il est question à l'aide de la métaphore du « canal bruité » [1] (Shannon 1948) : étant donné une séquence d'observations O, il faut trouver le message M le plus probable, c'est-à-dire celui qui maximise :

$$P(M,O) = P(O|M)P(M) \tag{4.1.0.1}$$

où $P(O|M)$ estime la *transduction* entre observations et messages, et $P(M)$ estime le langage. Notons que la transduction entre message et observations peut comprendre plusieurs étapes, correspondant à des niveaux de représentation :

$$\begin{aligned} &\textit{Etant donné } M = L_0 \textit{ et } O = L_k, \\ &\quad P(L_0, L_k) = P(L_k|L_0)P(L_0) \\ &\quad \textit{tel que } P(L_k|L_0) = \sum_{l_1,\cdots,l_{k-1}} \prod_{1\leq j\leq k} P(L_j|L_{j-1}) \end{aligned} \tag{4.1.0.2}$$

où l'on fait l'hypothèse que chaque niveau dans cette cascade est indépendant du précédent. En reconnaissance de la parole par exemple, les observations sont des vecteurs de caractéristiques acoustiques prises sur le signal, et le message est une suite de mots. Les niveaux intermédiaires pourraient dans ce cas être les phonèmes, les syllabes et les mots (Pereira

[1] Traduction de la notion anglaise *noisy channel*.

& Sproat 1994). Notons que pour des besoins d'implémentation, les sommes et les produits sont souvent remplacés par des minimisations et des sommes de logarithmes négatifs :

$$\begin{aligned} &\text{-}\log P(L_0, L_k) = \text{-}\log P(L_k|L_0) + \text{-}\log P(L_0) \\ &\textit{tel que } \ \text{-}\log P(L_k|L_0) \approx \min_{l_1,\cdots,l_{k-1}} \sum_{1\leq j\leq k} \text{-}\log P(L_j|L_{j-1}) \end{aligned} \tag{4.1.0.3}$$

Dans cette formulation, en faisant l'hypothèse que l'approximation est raisonnable, le message le plus probable est celui qui minimise $\text{-}\log(P(L_0, L_k))$.

Généralement, chaque transduction de cette cascade $\text{-}\log P(L_j|L_{j-1})$ ainsi que le langage $\text{-}\log P(L_0)$ sont modélisés *via* un outil à états finis, comme les modèles de Markov caché (HMM [2]). Cependant, comme le font remarquer les chercheurs (Pereira & Sproat 1994, Pereira & Riley 1997), les différents niveaux sont souvent modélisés à l'aide de moyens *ad hoc*, sans exploiter les similarités qu'ils partagent. Or, en ce qui concerne chaque étape de la cascade, on peut informellement considérer qu'il s'agit d'une projection d'une paire entrée-sortie (seq_1, seq_2) sur des probabilités $P(seq_2|seq_1)$. Plus formellement, chaque étape de la cascade est une *transduction pondérée* $T : \Sigma_1^* \times \Sigma_2^* \to \mathbb{K}$ où Σ_1^* et Σ_2^* sont les ensembles de strings définis sur Σ_1 et Σ_2, et $\mathbb{K}$ est un ensemble de poids, par exemple des probabilités entre 0 et 1. De même, le langage peut informellement être considéré comme une projection d'une séquence *seq* sur des probabilités $P(seq)$. Plus formellement, il s'agit d'un *langage pondéré* $A : \Sigma^* \to \mathbb{K}$ où Σ^* est l'ensemble de strings défini sur Σ.

Le point commun de ces systèmes est qu'ils mettent en œuvre la recherche du meilleur chemin dans un graphe orienté pondéré : l'idée est de déterminer le chemin le meilleur à partir d'un état donné vers tous les autres états du graphe. Dans ce contexte, les poids des transitions peuvent représenter des distances, des probabilités ou tout autre type de quantité que l'on peut sommer le long d'un chemin et que l'on désire minimiser. Les poids des transitions sont des nombres réels (ils appartiennent à $\mathbb{R}$) et les opérations utilisées sont l'addition le long d'un chemin et la recherche du minimum entre les différents chemins.

Ce qui est intéressant, c'est que les séries rationnelles et les semi-anneaux constituent un cadre de travail idéal pour généraliser cette notion de recherche du meilleur chemin à d'autres types de valeurs : les *poids* de l'ensemble $\mathbb{K}$ sur lequel la recherche du meilleur chemin sera réalisée peuvent être des nombres réels, mais également des strings, des expressions régulières ou toute autre quantité qui peut être *multipliée* le long d'un chemin à l'aide d'une opération $\otimes$ et qui peut être *sommée* à l'aide d'une opération $\oplus$. Le poids d'un chemin sera dès lors obtenu par multiplication des poids des transitions à l'aide de $\otimes$, et la distance la plus courte entre deux états sera calculée comme la somme des poids de tous les chemins entre ces états à l'aide de $\oplus$.

Forts de ce constat et sur la base des semi-anneaux et des séries entières rationnelles, Mohri *et al.* ont proposé des extensions aux algorithmes classiques définis sur les automates et les transducteurs (suppression-ϵ, déterminisation, minimisation, composition) applicables

[2] *Hidden Markov Model* en anglais.

aux automates et transducteurs pondérés, ainsi qu'un algorithme de recherche des n meilleurs chemins d'une machine pondérée. Ces extensions reposent toutes sur la notion de *distance la plus courte*.

Les notions mathématiques sur lesquelles reposent les algorithmes applicables aux machines pondérées sont évoquées dans (Pereira & Sproat 1994, Pereira & Riley 1997, Mohri 1996, Mohri *et al.* 1996, 2000, 2002) et décrites dans le détail dans (Mohri 2002b). Néanmoins, pour la clarté de l'exposé, cette section commence par en donner une présentation succincte avant de définir les machines pondérées. Sur cette base, la notion de *distance la plus courte* dans un graphe pondéré est définie. Les différents algorithmes qui reposent sur cette distance la plus courte sont ensuite présentés. Nous commençons par la recherche des n meilleurs chemins, étant donné que cet algorithme est le but avoué de l'emploi de machines pondérées. Nous terminons par les algorithmes d'optimisation des machines pondérées : suppression-ϵ, déterminisation et minimisation.

4.2 Fondements mathématiques

4.2.1 Semi-anneau

Un semi-anneau est une structure algébrique consistant en un ensemble d'éléments, muni des opérations *somme* $\oplus$ et *produit* $\otimes$ qui vérifient certaines des propriétés de la somme et du produit des nombres ordinaires. Plus formellement,

Définition 4.2.1. *Un* semi-anneau *est un 5-uplet* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ *tel que :*

- $\mathbb{K}$ *est un ensemble,*
- $(\mathbb{K}, \oplus, \overline{0})$ *est un monoïde commutatif dont* $\oplus$ *est la loi de composition interne associative et* $\overline{0}$*, l'élément neutre,*
- $(\mathbb{K}, \otimes, \overline{1})$ *est un monoïde dont* $\otimes$ *est la loi de composition interne associative et* $\overline{1}$*, l'élément neutre,*
- $\otimes$ *est distributive par rapport à* $\oplus$*,*
- $\overline{0}$ *est absorbant pour* $\otimes$ *:* $\forall x \in \mathbb{K} : x \otimes \overline{0} = \overline{0} \otimes x = \overline{0}$.

Notons que dans un anneau, $(\mathbb{K}, \oplus, \overline{0})$ est un *groupe*, c'est-à-dire un monoïde dont tous les éléments sont inversibles : $\forall x \in \mathbb{K}, \exists y \in \mathbb{K} : x \oplus y = y \oplus x = \overline{0}$. y est dit l'inverse de x et se note x^{-1}. Informellement, un semi-anneau est donc un anneau sur lequel l'inverse n'est pas obligatoirement défini.

Un semi-anneau est dit *commutatif* lorsque son produit est commutatif : $\forall a, b \in \mathbb{K}, a \otimes b = b \otimes a$.

Définition 4.2.2. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau. Un élément* $a \in \mathbb{K}$ *est* idempotent *si* $a \oplus a = a$. $\mathbb{K}$ *est dit* idempotent *si tous ses éléments sont idempotents.*

Lemme 4.2.1. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau idempotent. La relation* $\leq_{\mathbb{K}}$ *définie par* $(a \leq_{\mathbb{K}} b) \Leftrightarrow (a \oplus b = a)$ *définit un ordre partiel sur* $\mathbb{K}$ *appelé l'*ordre naturel sur $\mathbb{K}$.

Définition 4.2.3. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau, et* $\leq$ *un ordre partiel sur* $\mathbb{K}$*.* $\mathbb{K}$ *est* négatif *si* $\overline{1} \leq \overline{0}$*,* positif *si* $\overline{0} \leq \overline{1}$.

Définition 4.2.4. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau, et* $\leq$ *un ordre partiel sur* $\mathbb{K}$*. On dit que* $\mathbb{K}$ *est* monotonique *si* $\forall a, b, c \in \mathbb{K}$*,*

$$(a \leq b) \Rightarrow (a \oplus c \leq b \oplus c),$$
$$(a \leq b) \Rightarrow (a \otimes c \leq b \otimes c) \text{ et } (c \otimes a \leq c \otimes b).$$

L'idempotence combinée à la monotonicité et à la négativité rend l'ordre partiel sur $\mathbb{K}$ unique : $\forall a \in \mathbb{K}$, $\overline{1} \leq a \leq \overline{0}$.

Proposition 4.2.1. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau avec un ordre partiel* $\leq$*. Supposons que* $\mathbb{K}$ *soit négatif, idempotent et monotonique, alors* $\leq$ *correspond à l'ordre naturel* $\leq_{\mathbb{K}}$*.*

Définition 4.2.5. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau.* $\mathbb{K}$ *est* borné *si* $\overline{1}$ *est un annulateur pour* $\oplus$ *:* $\forall a \in \mathbb{K}$, $\overline{1} \oplus a = \overline{1}$.

Définition 4.2.6. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau borné. Alors,* $\mathbb{K}$ *est idempotent.*

Pour un semi-anneau borné $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ avec l'ordre naturel $\leq_{\mathbb{K}}$, nous avons $\forall a \in \mathbb{K}$, $\overline{1} \leq_{\mathbb{K}} a \leq_{\mathbb{K}} \overline{0}$. Ceci justifie le terme *borné*.

Les algorithmes de calcul de la distance la plus courte proposés par Mohri reposent sur la notion de semi-anneau k-clos.

Définition 4.2.7. *Soit un entier* $k \geq 0$ *et* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau commutatif.* $\mathbb{K}$ *est* k-clos *si :*

$$\forall a \in \mathbb{K}, \bigoplus_{n=0}^{k+1} a^n = \bigoplus_{n=0}^{k} a^n$$

Quand $k = 0$, l'expression peut être réécrite $\forall a \in \mathbb{K}, \overline{1} \oplus a = \overline{1}$. On constate donc que la notion de clôture 0 correspond à celle de borne pour les semi-anneaux commutatifs.

Lemme 4.2.2. *Soit* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, un semi-anneau* k*-clos, alors :*

$$\forall l > k, \forall a \in \mathbb{K}, \bigoplus_{n=0}^{l} a^n = \bigoplus_{n=0}^{k} a^n$$

4.2.2 Séries entières rationnelles

L'étude des séries entières formelles constitue un domaine majeur de l'informatique théorique (Eilenberg 1974, Kuich & Salomaa 1986). Une série entière formelle est un outil qui permet d'utiliser l'arsenal analytique des séries entières, sans tenir compte de la notion de convergence.

Vue comme une fonction, une série entière formelle $S : u \mapsto (S, u)$ est une fonction d'un ensemble E vers un semi-anneau $\mathbb{K}$: (S, u) est l'image de u dans S et est appelée le *coefficient* de u dans S. Les séries entières rationnelles sont exactement les séries entières formelles qui peuvent être construites par les opérations rationnelles (concaténation, union et étoile de Kleene) à partir de la série entière des *singletons* tel que $(S, u) = k$, $(S, v) = \overline{0}$ si $u \neq v$ pour $u \in \Sigma^*$, $k \in \mathbb{K}$. Vue comme une fonction, une série entière rationnelle $S : u \mapsto (S, u)$ est donc une fonction d'un monoïde libre Σ^* vers un semi-anneau $\mathbb{K}$.

Note 4.2.1. Les notations suivantes s'appliquent aux séries rationnelles :

- La notation $\sum_{u \in \Sigma^*} (S, u)u$ définit la série rationnelle par ses coefficients.
- Le *support* de S est le langage support de la série défini par :

$$supp(S) = \{u \in \Sigma^* : (S, u) \neq \infty\}$$

L'un des théorèmes majeurs dans la théorie des séries rationnelles est le théorème de Schützenberger (1961), qui généralise le théorème de Kleene (cf. Théorème 2.5.1) en établissant l'équivalence des séries rationnelles et des automates pondérés :

Théorème 4.2.1 (Schützenberger, 1961)**.** *Les séries entières rationnelles sont exactement ces séries entières formelles qui peuvent être représentées par des automates pondérés.*

En d'autres termes, une série formelle est rationnelle si et seulement si elle est reconnaissable, c'est-à-dire représentable par un automate pondéré.

4.3 Machines à états finis pondérées

Les machines à états finis pondérées sont une généralisation de la notion de machines à états finis : chaque transition d'un machine pondérée reçoit un poids en plus de l'étiquetage habituel.

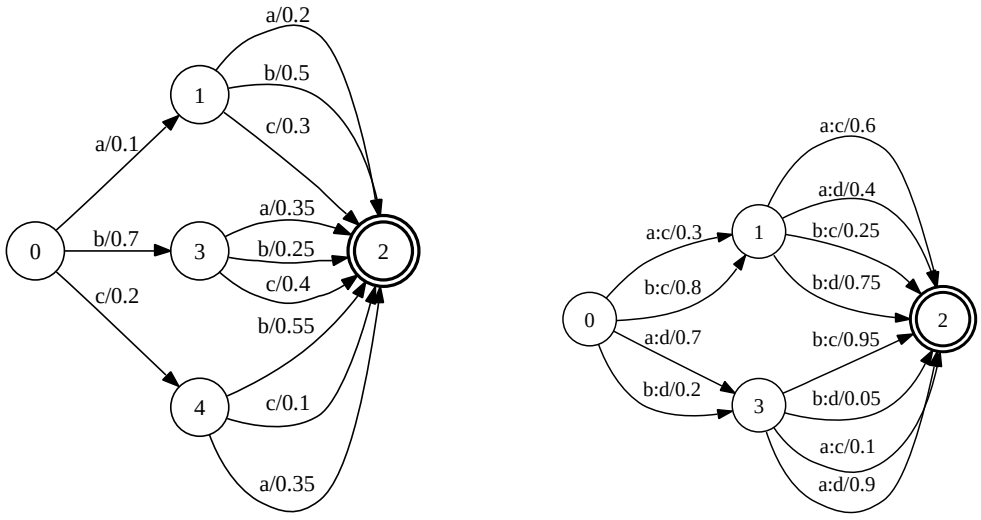
Définition 4.3.1. *Un* automate à états finis pondéré A_K *sur un semi-anneau* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ *est un 6-uplet* $(\Sigma, Q, i, F, E, \rho)$ *où* Σ *est un ensemble fini de symboles appelé alphabet,* Q *est un ensemble fini d'états,* $i \in Q$ *est l'état initial,* $F \subseteq Q$ *est l'ensemble des états finaux,* $E \subseteq Q \times \Sigma \times \mathbb{K} \times Q$ *est un ensemble fini de transitions* e *de la forme* (p, a, w, q) *et* $\rho : F \to \mathbb{K}$ *est la fonction de pondération finale.*

Un automate pondéré A_K (cf. Figure 4.1, (a)) définit donc une série entière rationnelle $S(A_K) = \Sigma^* \mapsto \mathbb{K}$.

Définition 4.3.2. *Un* transducteurs à états finis pondéré T_K *sur le semi-anneau* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ *est un 7-uplet* $(\Sigma_1, \Sigma_2, Q, i, F, E, \rho)$ *où* Σ_1 *est un ensemble fini de symboles appelé alphabet d'entrée,* Σ_2 *est un ensemble fini de symboles appelé alphabet de sortie,* Q*,* i*,* F *et* ρ *sont définis comme dans un automate pondéré, et* $E \subseteq Q \times \Sigma_1 \times \Sigma_2 \times \mathbb{K} \times Q$ *est un ensemble fini de transitions e de la forme* (p, a_1, a_2, w, q)*.*

Un transducteur pondéré T_K (cf. Figure 4.1, (b)) définit donc une série entière rationnelle $S(T_K)$, si ce n'est que $S(T_K)$ est une série rationnelle d'un monoïde libre vers un semi-anneau de séries rationnelles : $\Sigma_1^* \mapsto (\Sigma_2^* \otimes \mathbb{K})$.

Note 4.3.1. Les machines pondérées présentent parfois une fonction $\lambda : I \to \mathbb{K}$, qui est une fonction de pondération initiale. Nous n'avons pas inclus cette fonction dans la définition des machines, parce que les machines présentées n'ont pas un ensemble I d'états initiaux, mais un seul état initial i. Tous les chemins d'une machine partant d'une même source, la pondération initiale ne permet pas de les distinguer, et devient inutile.



(a) Automate A_K (b) Transducteur T_K

FIG. 4.1: Machines à états finis pondérées

4.4 Problème de la distance la plus courte

Dans cette section, nous employons la notion de *graphe orienté* afin de traiter de manière indifférenciée les automates et les transducteurs. Etant donné $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$, un semi-anneau et $G_K = (Q, i, F, E, \rho)$, un graphe orienté pondéré sur $\mathbb{K}$, un *chemin* $\pi = e_1 e_2 \dots e_{k-1} e_k$ est un élément de E^* tel que :

- $e_j.q = e_{j+1}.p$ pour $j = 1, \cdots, k-1$,
- $\pi.p = e_1.p$,
- $\pi.q = e_k.q$.

On peut définir w, une fonction de pondération d'un chemin qui calcule le poids d'un chemin comme le résultat du $\otimes$-produit des poids de ses transitions :

$$w(\pi) = \bigotimes_{j=1}^{k} e_j.w$$

Un ensemble de chemins allant de p à q est noté $P(q)$. Etant donné que $\oplus$ est commutative dans $\mathbb{K}$, le poids associé à l'ensemble des chemins $P(q)$ correspond à :

$$w(P(q)) = \bigoplus_{\pi \in P(q)} w(\pi)$$

Soit $s \in Q$, un état fixé comme étant la *source*. Pour tout état $q \in Q$, $D(s, q)$ note *la distance la plus courte* de s à q dans le graphe orienté pondéré G_K et se définit :

$$\begin{cases} D(s, s) = \overline{1} \\ \forall q \in Q - \{s\}, D(s, q) = w(P(q)) \end{cases} \tag{4.4.0.1}$$

En considérant que π est un cycle si $\pi.p = \pi.q$, on peut étendre la définition d'un semi-anneau k-clos (Définition 4.2.7) comme suit :

Définition 4.4.1. *Soit un entier $k \geq 0$, $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ un semi-anneau commutatif et $G_K = (Q, i, F, E, \rho)$ un graphe orienté pondéré sur $\mathbb{K}$.*
$(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ est k-clos pour G_K si, pour n'importe quel cycle π de G_K :

$$\bigoplus_{n=0}^{k+1} w(\pi)^n = \bigoplus_{n=0}^{k} w(\pi)^n$$

Pour tout état $q \in Q$, $P_k(q)$ est l'ensemble fini des chemins de s à q avec au plus k occurrences d'un cycle. Quand $k = 0$, $P_k(q)$ est l'ensemble des chemins simples de s à q. Grâce au Lemme 4.2.2, pour un semi-anneau $\mathbb{K}$ k-clos pour G_K, nous avons :

$$\forall l > k, \bigoplus_{\pi \in P_l(q)} w(\pi) = \bigoplus_{\pi \in P(q)} w(\pi)$$

Puisque $w(P_\infty(q)) = w(P(q))$, on peut définir :

$$\bigoplus_{\pi \in P(q)} w(\pi) = \bigoplus_{\pi \in P_k(q)} w(\pi)$$

et donc définir la plus courte distance $D(s, q)$ pour tout $q \in Q$ quand $\mathbb{K}$ est k-clos pour G_K.

Parmi les semi-anneaux commutatifs, idempotents, négatifs, monotoniques, bornés et k-clos, on trouve entre autres $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$, appelé le *semi-anneau tropical*, qui est la structure algébrique de la plupart des algorithmes classiques de recherche des meilleurs chemins (cf. Equation 4.1.0.3). Tout graphe G_K défini sur ce semi-anneau ne possède pas de cycle négatif, ce qui permet d'affirmer que pour tout cycle c

$$\min(w(c), 0) = 0$$

Cette notion de la distance la plus courte est la base des algorithmes proposés par Mohri *et al.* pour la recherche des n meilleurs chemins d'un graphe et pour l'optimisation des graphes (suppression-ϵ, déterminisation, minimisation).

4.5 Recherche des n meilleurs chemins d'un graphe

Etant donné un automate pondéré $A_K = (\Sigma, Q, i, F, E, \rho)$ défini sur le semi-anneau tropical, Mohri & Riley (2002) ont proposé un algorithme efficace de recherche des n meilleurs chemins. Cet algorithme se déroule en deux étapes. La première étape consiste à calculer la distance la plus courte entre chaque état $q \in Q$ et F. Sur cette base, la seconde étape consiste à rechercher effectivement les n meilleurs.

4.5.1 Première étape

Calculer la distance la plus courte entre un état $p \in Q$ et F revient à résoudre :

$$\phi(p) = \min\{w(\pi) + \rho(f) : \pi \in P(p, f), f \in F\} \tag{4.5.1.1}$$

Si l'on inverse le graphe orienté et que les états de F sont réunis en un seul état initial, le calcul de la distance la plus courte entre $q \in Q$ et F revient à résoudre la recherche du plus court chemin à partir d'une source i unique. Dans ce contexte, l'algorithme de Dijkstra est tout indiqué (Aho *et al.* 1974). En voici le principe.

Dijkstra. L'idée est de construire un ensemble d'états Q' correspondant à Q, mais dans lequel les états sont ajoutés dans l'ordre croissant de leur distance par rapport à la source i. Les distances des différents états sont mémorisées dans un vecteur V_{min} mis à jour en cours de processus. L'algorithme initialise $V_{min}[p]$ pour tout $p \in Q$ par la distance la plus

courte d'une transition directe entre i et p : $D(i,p)$. Si aucune transition directe n'existe entre les deux états, $V_{min}[p]$ vaut alors $+\infty$. L'algorithme ajoute ensuite itérativement dans Q' l'état p tel que $V_{min}[p]$ est un minimum, et en profite pour réévaluer la distance des états q atteints à partir de p afin de mettre leur valeur $V_{min}[q]$ à jour si nécessaire : $V_{min}[q] = \min(V_{min}[q], V_{min}[p] + D(p,q))$.

Dans l'implémentation la plus simple de l'algorithme de Dijkstra, la recherche du minimum est une simple recherche linéaire parmi les états de Q (Aho *et al.* 1974), ce qui représente une complexité $O(|E| + |Q|^2)$. Cependant, avec une file de priorité implémentée sous la forme d'un tas de Fibonacci (Fredman & Tarjan 1987) et à l'aide d'une liste d'adjacence des états, la complexité est réduite à $O(|E| + |Q| \log |Q|)$ (Mohri & Riley 2002). Ce gain en terme de complexité requiert quelques explications.

Tas de Fibonacci. Un tas (Cormen *et al.* 1990) est une structure arborescente d'éléments pourvus d'une clef sur laquelle est définie une propriété d'ordre : étant donné un tas T et $a, b \in T$, si b est un fils de a, alors $clef(a) \leq clef(b)$ de sorte que la racine de T contient toujours l'élément du tas qui a la plus petite clef [3]. Cette propriété d'ordre définie sur les tas explique que cette structure de données soit couramment employée pour implémenter des files de priorité. Les opérations qui sont définies sur le tas sont :

- `INSERT` : ajout d'une clef (et donc d'un élément) dans le tas.
- `GETMIN` : renvoi de la racine du tas.
- `DELMIN` : suppression de la racine du tas.
- `EXTRACTMIN` : renvoi et suppression de la racine du tas.
- `DEL` : suppression d'un élément du tas.
- `DECREASE` : mise à jour de la valeur d'une clef pour autant que la nouvelle valeur soit inférieure à l'ancienne.
- `MERGE` : réunion de deux tas en un seul.

A la base du tas de Fibonacci se trouve le tas binomial (cf. Figure 4.2, (a)) qui est un tas d'arbres binomiaux définis récursivement comme suit :

- Un arbre binomial d'ordre 0 possède un seul nœud.
- Un arbre binomial d'ordre n possède 2^n nœuds et ses fils sont, dans l'ordre, des arbres binomiaux d'ordre $n-1, n-2, \ldots, 1, 0$.

Sur un tas binomial est définie une opération de fusion de deux tas qui assure la création d'un arbre binomial et respecte la propriété d'ordre du tas. Lors de la fusion, s'il existe un

[3] La définition que nous donnons est en fait celle d'un tas *minimal*. Le tas *maximal* existe également (il contient toujours en racine l'élément qui a la plus grande clef), mais ce type de tas n'intéresse pas notre propos.

seul arbre d'ordre j, celui-ci est ajouté au tas fusionné. Par contre, si deux arbres d'ordre j existent, ils sont fusionnés en un arbre d'ordre $j+1$. Ce processus est récursif de sorte que l'algorithme peut avoir à examiner trois arbres de chaque ordre : deux provenant des tas initiaux, et un du nouveau tas. Pour un tas de n nœuds, la complexité de la fusion est donc $O(\log n)$.

Le tas de Fibonacci possède un meilleur temps d'exécution amorti que le tas binomial, parce que la fusion n'est pas réalisée après chaque opération. L'avantage du tas de Fibonacci est dû à sa structure de représentation (cf. Figure 4.2, (b)) :

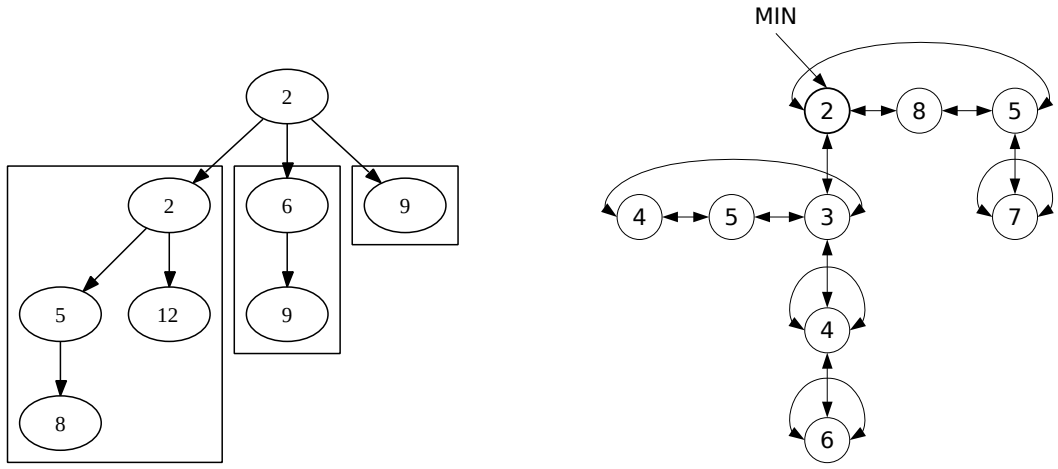
- les nœuds racines sont représentés à l'aide d'une liste doublement chaînée circulaire, de même que les nœuds fils d'un nœud donné.
- Chaque nœud connaît son nombre de fils et possède un marquage éventuel.
- Un pointeur indique le nœud contenant la clef minimale.

En fait, dans un tas de Fibonacci, on distingue deux groupes d'opérations : les opérations `GETMIN`, `MERGE`, `INSERT` et `DECREASE` (groupe 1) ont un temps constant :

- `GETMIN` est immédiat grâce au pointeur sur le minimum.
- `MERGE` consiste simplement à concaténer les racines de deux arbres et à comparer leurs racines pour mettre à jour le pointeur du nœud minimal. L'opération s'arrête là, puisqu'elle ne remet pas en cause l'ordre interne des arbres à réunir.
- `INSERT` crée un arbre d'ordre 0 avec l'élément à insérer, et le réunit à l'aide d'un `MERGE` avec la racine du tas initial.
- `DECREASE` diminue la clef d'un nœud. Si la propriété de tas est violée, le nœud est détaché de son père. Si le père est racine, le processus s'interrompt. Sinon, s'il n'est pas marqué, il est marqué et le processus s'interrompt. Sinon, le père est lui-même séparé de son père sur lequel les mêmes opérations sont réalisées, jusqu'à ce que la racine ou un nœud non marqué soit atteint. L'opération crée donc k nouveaux arbres dont les racines, si elles étaient marquées, perdent leur marquage. Un seul nœud peut donc être marqué après cette opération. La complexité de l'opération est donc $O(k)$, et le temps amorti est constant.

Les opérations `DELMIN`, `DEL` et `EXTRACTMIN` (groupe 2) sont de complexité $O(\log n)$.

- `DELMIN` supprime la racine du tas, de sorte que ses fils deviennent les racines de nouveaux arbres. C'est ici que l'opération de fusion définie sur les tas binomiaux est exécutée. La complexité est donc bien $O(\log n)$.
- `DEL` diminue la clef du nœud à supprimer en lui donnant la plus petite valeur possible, $-\infty$. Le nœud est ensuite supprimé en appliquant `DELMIN`.
- `EXTRACTMIN` exécute `GETMIN` suivie de `DELMIN`.



(a) Tas binomial d'ordre 3 (b) Tas de Fibonacci

FIG. 4.2: Tas binomial et tas de Fibonacci

En d'autres termes, à partir d'une structure vide, toute séquence de a opérations du groupe 1 et de b opérations du groupe 2 prendra un temps $O(a + b \log n)$, alors que dans un tas binomial, la même suite d'opérations prendrait $O((a+b) \log n)$. Le tas de Fibonacci est donc très intéressant pour autant que b soit asymptotiquement plus petit que a.

Algorithme. Pour un automate $A_K = (\Sigma, Q, i, F, E, \rho)$ défini sur le semi-anneau $\mathbb{K}$, le Pseudocode 16 présente l'algorithme qui calcule la distance la plus courte entre tout état $p \in Q$ et F. Le vecteur V_{min} qui contiendra les distances est initialisé à 0 pour tout état $p \in Q$ (ligne 1). Un graphe G_r est ensuite créé (ligne 2) en faisant appel à la procédure `REV-MIN-ADJACENT` dont l'algorithme est présenté en Pseudocode 17. Ce graphe est l'inverse de la liste d'adjacence de Q dans A_K, où chaque transition entre deux états q et p présente le minimum de toutes les transitions allant de p à q dans A_K : $D(p, q)$ (cf. Figure 4.3).

Au lieu de constituer un nouvel ensemble Q' à partir de Q comme évoqué dans l'explication du principe, on compte simplement le nombre d'états de Q dont la distance a déjà été calculée (ligne 14).

Le classement des états est réalisé à l'aide du tas de Fibonacci T, initialisé (lignes 4–10) par les transitions directes partant de i_r, la source de G_r (cf. Pseudocode 17). Ensuite, à chaque itération, l'état p qui a la distance la plus courte depuis la source est extrait (ligne 12) et placé dans le vecteur des distances (ligne 13). Enfin, la distance des états restants qui ont une transition qui quitte p est mise à jour si nécessaire (ligne 15). L'utilisation d'un tas de Fibonacci sera véritablement rentable si le nombre d'`EXTRACTMIN` (ligne 12) est asymptotiquement plus petit que le nombre d'`INSERT` et de `DECREASE` (lignes 6, 8 et 15).

4.5.2 Seconde étape

La seconde étape construit les n meilleurs chemins de A_K. Notons que le Pseudocode 18, qui en présente l'algorithme, intègre la première étape décrite ci-dessus (ligne 1). L'algorithme est une généralisation de celui de Dijkstra : il construit les n meilleurs chemins à partir de l'état initial en utilisant une file de priorité (un tas de Fibonacci) dont on extrait l'état p de coût minimal c à chaque itération (ligne 8). Le processus itératif s'arrête lorsque l'état final de A_K a été extrait n fois (ligne 12). Cette condition de terminaison suppose donc que A_K possède un seul état final, raison pour laquelle l'automate est modifié en ajoutant un seul état final, auquel les anciens états finaux sont reliés par des transitions ϵ (ligne 3).

On remarque que tout couple (p, c) est adjoint d'un élément π (ligne 8). Il s'agit d'un nœud *prédécesseur* qui permet de construire les n meilleurs chemins sous la forme de listes chaînées. Lorsqu'un triplet (p, c, π) est non final (lignes 15–18), pour chaque transition (p, a, w, q), un nouveau triplet (q, c', π') est créé et inséré dans la file. Le nœud π' vaut (a, w, π), et le coût c' est égal à :

$$c - V_{min}[p] + w + V_{min}[q] \tag{4.5.2.1}$$

En incluant $c - V_{min}[p]$ (ligne 10), ce coût permet de tenir compte du coût de la transition à partir de laquelle l'état p a été atteint.

Lorsqu'un triplet (p, c, π) est final (ligne 20), son nœud π est ajouté à la liste des chemins. Les chemins seront donc construits (ligne 24) à partir de leurs états finaux en remontant jusqu'à l'état initial [4].

Recherche déterministe des n meilleurs chemins. L'algorithme original de Mohri & Riley (2002) réalise la deuxième étape sur un graphe déterministe de l'automate pondéré A_K. L'avantage évident, dans ce cas, est que les n meilleurs chemins seront effectivement *différents*, puisque les hypothèses redondantes auront été supprimées avant plutôt qu'après la sélection des n meilleurs chemins.

Ceci suppose d'insérer un appel à un algorithme de déterminisation entre les lignes 1 et 2 du Pseudocode 18. Cependant, Mohri & Riley ne déterminisent pas la totalité de l'automate ; ils réalisent une déterminisation *au vol* de la partie de l'automate effectivement visitée au cours de la construction des n meilleurs chemins. La phase de déterminisation est donc réduite à son strict nécessaire, puisque seuls les états visités sont déterminisés. On parle d'évaluation retardée ou paresseuse.

Le principe général de l'algorithme de déterminisation est décrit en Section 4.6.2.

[4] La procédure `PATH2FSM` sera plus efficace si les listes de nœuds à parcourir sont doublement chaînées : une fois le nœud initial atteint, le graphe correspondant au chemin pourra être construit avec des états croissants en remontant les nœuds jusqu'à l'état final.

Require: Un WFSA $A_K = (\Sigma, Q, i, F, E, \rho)$ sur le semi-anneau $\mathbb{K}$
Ensure: Le vecteur V_{min}, un vecteur de longueur $|Q| + 1$, chaque position p contenant le coût minimal de p à F

```
1: V_min[p] ← 0 for p ← 0 to |Q|
2: G_r ← REV-MIN-ADJACENT(A_K)
3: seen ← 0 ; T ← ∅
4: for each p ∈ Q do
5:    if ∃e ∈ E_r[i_r] : e.q = p then
6:       FIBO_INSERT(T, (p, e.w))
7:    else
8:       FIBO_INSERT(T, (p, +∞))
9:    end if
10: end for
11: while seen ≠ |Q| do
12:    (p, c) ← FIBO_EXTRACTMIN(T)
13:    V_min[p] ← c
14:    seen ← seen + 1
15:    for each e ∈ E_r[p] do FIBO_DECREASE(T, (e.q, c + e.w)) end for
16: end while
17: V_min[|Q|] ← 0
18: return V_min
```

Pseudocode 16: `BESTFROMFINAL`

Require: Un WFSA $A_K = (\Sigma, Q, i, F, E, \rho)$ sur le semi-anneau $\mathbb{K}$
Ensure: Le graphe $G_r = (Q_r, i_r, E_r)$, l'inverse de la liste d'adjacence de Q dans A_K. Dans G_r, chaque transition de q à p vaut $D(p, q)$ dans A_K

```
1: i_r ← |Q|
2: Q_r ← Q ∪ {i_r}
3: E_r ← ∅
4: for each p ∈ Q do
5:    for each e ∈ E[p] do
6:       if ∃e_r ∈ E_r : e_r.p = e.q, e_r.q = e.p then
7:          e_r.w ← min(e_r.w, e.w)
8:       else
9:          E_r ← E_r ∪ {(e.q, e.w, e.p)}
10:      end if
11:   end for
12:   E_r ← E_r ∪ {(i_r, ρ(p), p)} if p ∈ F
13: end for
14: G_r ←(Q_r, i_r, E_r)
15: return G_r
```

Pseudocode 17: `REV-MIN-ADJACENT`

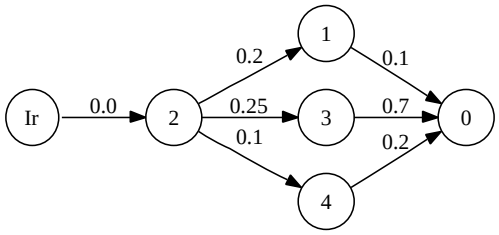


FIG. 4.3: Graphe G_r, inverse de la liste d'adjacence de l'automate A_K (cf. Figure 4.1, (a)), obtenu à l'aide de la procédure `REV-MIN-ADJACENT`. Dans G_r, le poids de q à p correspond à $D(p, q)$

Require: Un WFSA $A_K = (\Sigma, Q, i, F, E, \rho)$ sur le semi-anneau $\mathbb{K}$
Ensure: Le WFSA $A_{best} = (\Sigma, Q', i', F', E')$ contenant les n meilleurs chemins de A_K

$V_{min} \leftarrow$ `BESTFROMFINAL`(A_K)
$V_{seen}[p] \leftarrow 0$ **for** $p \leftarrow 0$ **to** $|Q|$
$f \leftarrow$ `JOINFINAL`(A_K)
$Path \leftarrow \emptyset$
$T \leftarrow \{(i, V_{min}[i], \texttt{NULL})\}$
$done \leftarrow$ `FALSE`
while $T \neq \emptyset$ **and** $done =$ `FALSE` **do**
  $(p, c, \pi) \leftarrow$ `FIBO_EXTRACTMIN`(T)
  $V_{seen}[p] \leftarrow V_{seen}[p] + 1$
  $c \leftarrow c - V_{min}[p]$
  if $V_{seen}[f] = n$ **then**
    $done \leftarrow$ `TRUE`
  else if $V_{seen}[p] < n$ **then**
    if $p \neq f$ **then**
      for each $e \in E[p]$ **do**
        $\pi' \leftarrow (e.a, e.w, \pi)$
        `FIBO_INSERT`$(T, (e.q, c + e.w + V_{min}[e.q], \pi'))$
      end for
    else
      `ADDPATH`$(Path, \pi)$
    end if
  end if
end while
$A_{best} \leftarrow$ `PATH2FSM`$(Path)$
return A_{best}

Pseudocode 18: `NBEST`

4.6 Optimisation des machines pondérées

4.6.1 Suppression-ϵ pondérée

Dans un automate pondéré avec transitions ϵ, $A_K = (\Sigma, Q, i, F, E, \rho)$, défini sur un semi-anneau $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$, l'ensemble de transitions est $E \subseteq Q \times \{\Sigma \cup \epsilon\} \times \mathbb{K} \times Q$.

Dans un transducteur pondéré avec transitions ϵ, $T_K = (\Sigma_1, \Sigma_2, Q, i, F, E, \rho)$, défini sur un semi-anneau $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$, l'ensemble de transitions est $E \subseteq Q \times \{\Sigma_1 \cup \epsilon\} \times \{\Sigma_2 \cup \epsilon\} \times \mathbb{K} \times Q$. Vu comme un automate *via* son automate sous-jacent, la transition ϵ de T_K est le couple des symboles entrée/sortie (ϵ,ϵ).

Comme nous l'avons noté précédemment (cf. Section 2.3.3), les transitions ϵ induisent un retard lors du parcours d'une string d'entrée dans le graphe orienté. Il est donc nécessaire de les supprimer.

L'algorithme de suppression-ϵ est classiquement présenté en combinaison avec la déterminisation, de sorte que l'ϵ-NFA est directement converti en DFA (Aho *et al.* 1986). Cependant, dans le cas de machines pondérées, la déterminisation n'est pas toujours calculable (cf. Section 4.6.2). L'algorithme proposé par Mohri (2002a) est donc présenté seul, et s'applique à tout graphe orienté pondéré avec transitions ϵ $G_K = (Q, i, F, E, \rho)$ défini sur un semi-anneau k-clos. Cet algorithme est présenté sur un automate, mais s'applique également à un transducteur *via* son automate sous-jacent.

Les étapes de l'algorithme sont les mêmes que celles décrites dans le cas des automates non pondérés (cf. Pseudocode 3) : la première étape est donc le calcul de la clôture-ϵ, et la seconde étape s'appuie sur la clôture-ϵ pour réaliser la suppression-ϵ.

Clôture-ϵ pondérée. Si on restreint A_K à A_{K_ϵ} en supprimant de A_K toutes les transitions non étiquetées ϵ, et étant donné deux états $p, q \in Q$, la distance-ϵ de p à q dans A_{K_ϵ} se note $D_\epsilon(p, q)$ et est définie par :

$$D_\epsilon(p, q) = \bigoplus_{\pi \in P(q), \pi.u = \epsilon} w(\pi)$$

où $\pi.u$ note la concaténation des étiquettes le long du chemin menant de p à q :

$$\pi.u = e_1.a \cdot e_2.a \ldots e_{n-1}.a \cdot e_n.a, \text{ avec } e_1.p = p,\ e_n.q = q$$

La distance $D_\epsilon(p, q)$ est bien définie pour toute paire d'états (p, q) de A_K quand $\mathbb{K}$ est un semi-anneau k-clos pour A_{K_ϵ}. Par définition, pour tout état p, $D_\epsilon(p, p) = \overline{1}$.

Sur cette base, le calcul de la clôture-ϵ d'un état p de l'automate A_K n'est plus un ensemble d'états comme dans la clôture-ϵ classique, mais un ensemble de paires (état, poids) :

$$Q_\epsilon[p] = \{(q, w) : q \in E_\epsilon[p],\ w \in \mathbb{K} - \overline{0} = D_\epsilon[p, q]\}$$

où $E_\epsilon[p]$ représente l'ensemble des états accessibles depuis p *via* des chemins uniquement étiquetés par ϵ.

Le Pseudocode 19 présente la clôture-ϵ sur le semi-anneau tropical, dans lequel :

(ligne 2) $D_\epsilon(p,p) = 0$

(lignes 5-11) $D_\epsilon(p,q) = \min_{\pi \in P(q), \pi.u = \epsilon} w(\pi)$ ssi $p \neq q$

Require: Q_ϵ, un vecteur d'ensembles de taille $|Q|$;
p, un état de Q
Ensure: $Q_\epsilon[p]$ contient la clôture-ϵ pondérée de p

```
1: if Q_ε[p] = ∅ then
2:   Q_ε[p] ←{(p, 0)}
3:   for each e ∈ E[p] : e.a = ε do
4:     W-CLOTURE-EPS(Q_ε, e.q)
5:     for each (q, w') ∈ Q_ε[e.q] do
6:       if ∃(q, w) ∈ Q_ε[p] then
7:         w ← min(w, e.w + w')
8:       else
9:         Q_ε[p] ← Q_ε[p] ∪ {(q, w')}
10:      end if
11:    end for
12:  end for
13: end if
```

Pseudocode 19: `W-CLOTURE-EPS`

Modification des transitions. Comme dans le cas de la suppression-ϵ classique (cf. Pseudocode 3), la seconde étape consiste à modifier l'ensemble de transitions $E[p]$ de tout état $p \in Q$:

- en supprimant les transitions ϵ de $E[p]$
- en ajoutant à $E[p]$ les transitions non-ϵ quittant les états appartenant à $Q_\epsilon[p]$.

L'algorithme pondéré diverge de l'algorithme classique dans le sens où toute transition e quittant l'état q d'un couple $(q, w') \in Q_\epsilon[p]$ est repondérée comme ceci :

$$e.w = e.w \otimes w'$$

En outre, lorsque $Q_\epsilon[p]$ contient un ou plusieurs états finaux, p devient final et la fonction ρ lui associe un poids défini comme suit :

$$\rho(p) = \bigoplus_{q \in Q_\epsilon[p] \cap F} D_\epsilon(p,q) \otimes \rho(q)$$

Le Pseudocode 20 présente l'algorithme pondéré sur le semi-anneau tropical, où le poids d'une transition est défini (ligne 22) par :

$$e.w = e.w + w'$$

et le poids d'un état final est défini (lignes 12–16) par :

$$\rho(p) = \min_{q \in Q_\epsilon[p] \cap F} D_\epsilon(p, q) + \rho(q)$$

Require: Un ϵ-WNFA $A_K = (\Sigma, Q, i, F, E, \rho)$
Ensure: Le WNFA $A_{K_N} = (\Sigma, Q', i', F', E', \rho')$ équivalent à A_K

1: $Q_\epsilon[p'] \leftarrow \emptyset$ **for** $p' \leftarrow 0$ **to** $|Q| - 1$
2: `W-CLOTURE-EPS`(Q_ϵ, p') **for** $p' \leftarrow 0$ **to** $|Q| - 1$
3: $H \leftarrow F' \leftarrow E' \leftarrow \emptyset$
4: $p' \leftarrow i' \leftarrow$ `INSERT`$(H, Q_\epsilon[i])$ /* 1^{er} élément $\rightarrow$ retour $= 0$ */
5: $Q' \leftarrow \{i'\}$
6: $\rho'[p] \leftarrow$ `UNDEF` **for** $p \leftarrow 0$ **to** $|Q| - 1$
7: **while** $p' < |Q'|$ **do**
8: $S_\epsilon \leftarrow$ `GET`(H, p')
9: $F_\epsilon \leftarrow \{(p, w) \in S_{p'} : p \in F\}$
10: **if** $F_\epsilon \neq \emptyset$ **then**
11: $F' \leftarrow F' \cup \{p'\}$
12: **for each** $(p, w) \in F_\epsilon$ **do**
13: **if** $\rho'(p') > (w + \rho(p))$ **or** $\rho'(p') =$ `UNDEF` **then**
14: $\rho'[p'] \leftarrow w + \rho(p)$
15: **end if**
16: **end for**
17: **end if**
18: **for each** $(p, w) \in S_\epsilon$ **do**
19: **for each** $e \in E[p] : e.a \neq \epsilon$ **do**
20: $q' \leftarrow$ `INSERT`$(H, Q_\epsilon[e.q])$
21: $Q' \leftarrow Q' \cup \{q'\}$
22: $E' \leftarrow E' \cup \{(p', e.a, w + e.w, q')\}$
23: **end for**
24: **end for**
25: $p' = p' + 1$
26: **end while**
27: $A_{K_N} \leftarrow (\Sigma, Q', i', F', E', \rho)$
28: **return** $_{K_N}$

Pseudocode 20: `W-SUPPRESSION-EPS`

4.6.2 Déterminisation pondérée

Tout comme l'automate non pondéré, un automate pondéré A_K est dit *déterministe* si et seulement si chaque état de A_K présente au plus une transition étiquetée par un symbole donné de l'alphabet. La Figure 4.4 donne un exemple d'automate pondéré non-déterministe.

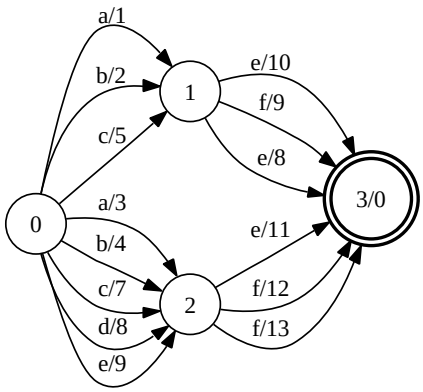


FIG. 4.4: Automate pondéré non déterministe A_{K_1}, repris de (Mohri & Riley 1999)

La déterminisation pondérée s'applique à un automate pondéré et génère un automate pondéré équivalent qui est déterministe. Cependant, contrairement aux automates non pondérés, les automates pondérés ne peuvent pas tous être déterminisés. Mohri (1997a) a démontré que tous les automates qui ont la propriété de gémellité sont déterminisables.

Définition 4.6.1. *Etant donné un automate pondéré* $A_K = (\Sigma, Q, i, F, E, \rho)$ *sur un semi-anneau* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$*, deux états* $p, q \in Q$ *sont dits* jumeaux *si :*

$$\forall (u, v) \in (\Sigma^*)^2, (\{p, q\} \subset \hat{d}(i, u),\ p \in \hat{d}(p, v),\ q \in \hat{d}(q, v)) \\ \Rightarrow w(p, v, p) = w(q, v, q)$$

où

$$w(p, x, q) = \bigoplus_{\pi \in (p,x,q)} w(\pi).$$

On dit que A_K a la propriété de gémellité lorsque n'importe quelle paire d'états (p, q) a la propriété de gémellité. En accord avec la définition, deux états qui n'ont pas de cycle avec la même string v sont jumeaux. En particulier, deux états qui n'appartiennent à aucun cycle sont nécessairement jumeaux. Donc, tout automate pondéré acyclique a la propriété de gémellité. Ce résultat est particulièrement intéressant pour les applications en traitement du langage naturel, où les données à analyser sont par essence acycliques.

Algorithme. Le Pseudocode 21 présente l'algorithme sur le semi-anneau tropical. Etant donné un WNFA $A_K = (\Sigma, Q, i, F, E, \rho)$, la déterminisation pondérée est une généralisation de la déterminisation classique, basée sur la construction de l'ensemble des parties de Q (cf. Pseudocode 1). Cependant, étant donné que les transitions quittant un état p donné et étiquetées avec un symbole a donné peuvent présenter des poids différents, la transition du nouvel état dans le WDFA ne peut conserver que le minimum de ces poids. Il est donc nécessaire de mémoriser les excédents de poids. Ceci explique que les états du WDFA sont construits comme des ensembles de couples (q, w), où q est l'état atteint par une transition quittant p étiquetée a, et w est l'excédent de poids de la transition.

Le WDFA construit est $A'_K = (\Sigma, Q', i', F', E', \rho')$. Son état initial, i', correspond au couple $\{(i, 0)\}$, où i est associé à un poids nul, puisqu'il n'y a pas encore d'excédent (lignes 2–3). Les autres états sont construits *de proche en proche* à partir de l'état initial (lignes 5–36).

Etant donné un état $S = \{(p, w)\} \in Q'$ et un symbole $a \in \Sigma$, on construit une nouvelle transition (S, a, w', T) où w' vaut (lignes 17–19) :

$$w' = \min_{(p,w)\in S}(w + \min_{e\in E[p]:e.a=a} e.w) \tag{4.6.2.1}$$

et $T = \{(q, w_2)\}$ vaut (lignes 20–28) :

$$T = \bigcup_{(p,w)\in S} \{(q, \min_{e\in E[p]:e.a=a, e.q=q} w + e.w - w')\} \tag{4.6.2.2}$$

Tout état $S = \{(p, w)\} \in Q'$ est final s'il contient au moins un couple (p, w) tel que p est final (lignes 7–9). Dans ce cas, le poids final attribué à S par ρ' vaut (lignes 10–12) :

$$\rho'(S) = \min_{(p,w)\in S:p\in F} w + \rho(p) \tag{4.6.2.3}$$

La Figure 4.5 montre le résultat de l'application de la déterminisation pondérée sur l'automate de la Figure 4.4.

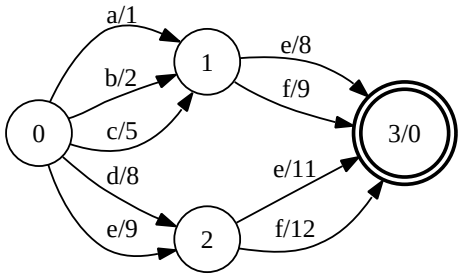


FIG. 4.5: Automate pondéré déterministe A_{K_2}, équivalent à A_{K_1}

Séquentialisation pondérée. Une analyse comparée des algorithmes de séquentialisation d'un transducteur (cf. Pseudocode 13) et de déterminisation pondérée d'un automate pondéré met en évidence l'identité des algorithmes, à ceci près qu'ils opèrent sur des *ensembles différents*. La déterminisation pondérée opère sur un ensemble de poids ($\mathbb{R}^+$ dans le cas du semi-anneau tropical), alors que la séquentialisation opère sur un ensemble de strings appartenant à un alphabet Σ_2.

La séquentialisation des transducteurs est un exemple du fait que les algorithmes basés sur le calcul de la distance la plus courte peuvent s'appliquer à d'autres valeurs que des poids : il peut s'agir de strings, pour autant que ces strings puissent être *multipliées* le long d'un chemin à l'aide d'une opération $\otimes$ et *sommées* à l'aide d'une opération $\oplus$. Le *poids* d'un chemin sera dès lors obtenu par multiplication des *poids* des transitions à l'aide de $\otimes$, et la distance la plus courte entre deux états sera calculée comme la somme des poids de tous les chemins entre ces états à l'aide de $\oplus$. Dans le cas de Σ_2^*, le produit est la *concaténation* et la somme est le *préfixe*, dans le semi-anneau $(\Sigma_2^* \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$, également k-clos (Mohri 2000). Les transducteurs non pondérés peuvent donc être considérés comme des séries rationnelles d'un monoïde libre Σ_1^* vers le semi-anneau de strings Σ_2^*.

L'équivalence précédente met également en évidence que les automates pondérés peuvent être considérés comme des transducteurs dont l'alphabet de sortie est l'ensemble des poids, par exemple $\mathbb{R}^+$ (Mohri 1997a).

Ce résultat est très important, car il permet de généraliser l'algorithme de déterminisation pondérée de manière à ce qu'il s'applique aux transducteurs pondérés, en considérant que le semi-anneau sur lequel opère l'algorithme est dans ce cas le $\otimes$-produit du semi-anneau de strings avec le semi-anneau de poids : $\Sigma_2^* \otimes \mathbb{K}$ (Mohri *et al.* 2000).

Dans ce cas, étant donné qu'un symbole d'entrée peut correspondre à plusieurs sorties différentes, la transition du nouveau transducteur ne peut contenir que le plus long préfixe commun, et il faut garder une trace des strings « excédentaires », comme dans le cas de la séquentialisation classique. La séquentialisation pondérée d'un transducteur opère dès lors sur des triplets (p, u, w) où p est un état du transducteur à séquentialiser, u est la string excédentaire et w est le poids excédentaire. L'algorithme, qui part de l'état initial $(i, \epsilon, 0)$, est exactement celui de la déterminisation pondérée, si ce n'est qu'il réalise également les opérations sur les strings présentes dans l'algorithme de séquentialisation.

4.6.3 Minimisation pondérée

Si l'on considère l'automate pondéré de la Figure 4.5 comme un automate non pondéré dont les symboles sont les paires (a, w) faites d'un symbole a et d'un poids w, il est évident que la minimisation classique n'aura aucun effet : tous les états de l'automate sont distinguables. Or, Mohri (1997a) a démontré qu'un algorithme de minimisation pondérée peut être appliqué aux automates pondérés définis sur le semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$, de manière à réduire leur nombre d'états au-delà de ce que la minimisation classique peut proposer. Par la suite, Eisner (2003) a proposé un algorithme de

Require: Un WNFA $A_K = (\Sigma, Q, i, F, E, \rho)$
Ensure: Le WDFA $A'_K = (\Sigma, Q', i', F', E', \rho')$ équivalent à A_K
1: $H \leftarrow F' \leftarrow E' \leftarrow \emptyset$
2: $p' \leftarrow i' \leftarrow \texttt{INSERT}(H, \{(i, 0)\})$ /* 1^{er} élément $\rightarrow$ retour $= 0$ */
3: $Q' \leftarrow \{i'\}$
4: $\rho'[p] \leftarrow \texttt{UNDEF}$ **for** $p \leftarrow 0$ **to** $|Q| - 1$
5: **while** $p' < |Q'|$ **do**
6: $S_{p'} \leftarrow \texttt{GET}(H, p')$
7: $F_{p'} \leftarrow \{(p, w) \in S_{p'} : p \in F\}$
8: **if** $F_{p'} \neq \emptyset$ **then**
9: $F' \leftarrow F' \cup \{p'\}$
10: **for each** $(p, w) \in F_{p'}$ **do**
11: $\rho'(p') \leftarrow w + \rho(p)$ **if** $\rho'(p') > (w + \rho(p))$ **or** $\rho'(p') = \texttt{UNDEF}$
12: **end for**
13: **end if**
14: **for each** $a \in \Sigma$ **do**
15: $S_{q'} \leftarrow \emptyset$
16: $w' \leftarrow +\infty$
17: **for each** $(p, w) \in S_{p'}$ **do**
18: $w' \leftarrow \min(w', w + e.w)$ **for each** $e \in E[p] : e.a = a$
19: **end for**
20: **for each** $(p, w) \in S_{p'}$ **do**
21: **for each** $e \in E[p] : e.a = a$ **do**
22: **if** $\exists (q, w_2) \in S_{q'} : q = e.q$ **then**
23: $w_2 \leftarrow \min(w_2, w + e.w - w')$
24: **else**
25: $S_{q'} \leftarrow S_{q'} \cup \{(e.q, w + e.w - w')\}$
26: **end if**
27: **end for**
28: **end for**
29: **if** $S_{q'} \neq \emptyset$ **then**
30: $q' \leftarrow \texttt{INSERT}(H, S_{q'})$
31: $Q' \leftarrow Q' \cup \{q'\}$
32: $E' \leftarrow E' \cup \{(p', a, w', q')\}$
33: **end if**
34: **end for**
35: $p' = p' + 1$
36: **end while**
37: $A'_K \leftarrow (\Sigma, Q', i', F', E', \rho')$
38: **return** A'_K

Pseudocode 21: `W-DETERMINISATION`

minimisation plus général, permettant de traiter des automates pondérés définis sur d'autres semi-anneaux, comme le semi-anneau réel $(\mathbb{R}, +, \times, 0, 1)$ ou son équivalent logarithmique.

4.6.3.1 Mohri (1997a)

Principe. Cet algorithme est une généralisation de la préfixation proposée par Mohri dans le contexte de la minimisation des transducteurs séquentiels, où la préfixation est appliquée sur des strings (cf. Section 3.4.3).

De même que le préfixe u^{-1} d'un sous-ensemble L de Σ^* définit un nouveau sous-ensemble

$$u^{-1}L = \{v : uv \in L\},$$

Mohri définit pour une série entière S une nouvelle série entière $u^{-1}S$:

$$u^{-1}S = \sum_{x \in \Sigma^*} (S, ux)x \tag{4.6.3.1}$$

Sur cette base, il définit la relation d'équivalence R_S sur Σ^* pour toute série entière sous-séquentielle telle que :

$$\begin{aligned} &\forall (u, v) \in \Sigma^*, u \, R_S \, v \iff \exists \, k \in \mathbb{R}, \\ &\quad (u^{-1} \, supp(S) = v^{-1} \, supp(S)) \quad et \\ &\quad ([u^{-1} \, S - v^{-1} \, S]_{/u^{-1} \, supp(S)} = k) \end{aligned} \tag{4.6.3.2}$$

En d'autres termes, la relation d'équivalence R_S définit k classes distinctes, chaque classe étant une série entière $u^{-1}S$. Sur cette base, Mohri démontre que l'automate pondéré minimal existe :

Théorème 4.6.1. *Toute fonction sous-séquentielle S est représentable par un automate pondéré minimal, dont le nombre d'états correspond à l'index de R_S.*

Cet automate calculera dans ce cas la série entière S' définie par :

$$\begin{aligned} &\forall u \in \Sigma^* : u^{-1} \, supp(S) = 0, (S', u) = 0 \\ &\forall u \in \Sigma^* : u^{-1} \, supp(S) \neq 0, (S', u) = \min_{x \in u^{-1} \, supp(S)} (S, ux) \end{aligned} \tag{4.6.3.3}$$

Algorithme. L'algorithme proposé par Mohri sur la base des classes d'équivalence R_S contient deux étapes : il applique la minimisation classique, mais seulement *après* une étape où les poids des transitions sont *poussés* de l'état final vers l'état initial. Cette étape, qui est en fait une préfixation de la pondération de l'automate, permet de mettre au jour l'équivalence de certains états de l'automate. La minimisation classique détecte ensuite les états équivalents de l'automate pondéré en considérant les paires (a, w) comme les symboles d'un automate non pondéré.

Pour un automate pondéré $A_K = (\Sigma, Q, i, F, d, w, \rho)$, l'algorithme qui pousse les poids, que nous appellerons *préfixation pondérée*, construit un nouvel automate $A'_K = (\Sigma, Q, i, F, d, w', \rho')$ qui ne diffère de A_K que par les poids associés à ses transitions et à ses états :

- $\forall (p, a) \in Q \times \Sigma,\ w'(p, a) = w(p, a) + D(d(p, a), F) - D(p, F)$;
- $\forall p \in Q,\ \rho'(p) = 0$.

où $D(p, F)$ est la distance la plus courte entre l'état p et l'ensemble des états finaux F telle que définie en Equation 4.4.0.1. En accord avec cette définition, nous avons :

$$\forall u \in \Sigma^* : \hat{d}(p, au) \in F,\ D(p, F) \leq w(p, a) + w(d(p, a), u) + \rho(\hat{d}(d(p, a)u))$$

ce qui est équivalent à :

$$D(p, F) \leq w(p, a) + D(d(p, a), F)$$

Donc, w' est correctement défini :

$$\forall (p, a) \in Q \times \Sigma,\ w'(p, a) \geq 0$$

et l'automate A'_K ne contient aucun poids négatif.

Le Pseudocode 22 présente l'algorithme de la préfixation pondérée. Etant donné que la définition de la fonction ρ' implique de connaître la distance la plus courte entre tout état $p \in Q$ et F, l'algorithme de préfixation pondérée s'exécute en deux étapes. La première étape (ligne 1) calcule la distance la plus courte entre tout état $p \in Q$ et F à l'aide de l'algorithme présenté en Section 4.5. La seconde étape met à 0 la fonction ρ' pour tous les états de Q (ligne 3), et calcule w' pour tout couple $(p, a) \in Q \times \Sigma$ (lignes 4–9).

Le temps de calcul nécessaire à la minimisation pondérée est fort réduit. Sa complexité est linéaire dans le cas d'un automate acyclique ($O(|Q| + |E|)$) et en $O(|E| \log |Q|)$ dans le cas contraire.

La Figure 4.6 montre le résultat de l'application de la préfixation pondérée sur l'automate de la Figure 4.5, et la Figure 4.7 montre le résultat de la minimisation classique appliquée sur l'automate de la Figure 4.6.

4.6.3.2 Eisner (2003)

L'algorithme d'Eisner (2003) repose sur la notion de *fonction suffixe*.

Définition 4.6.2 (Fonction suffixe)**.** *Etant donné un automate $A_K = (\Sigma, Q, i, F, E, \rho)$ défini sur le semi-anneau $\mathbb{K}$ et un état $p \in Q$, la fonction suffixe F_p de p est la fonction définie par A_K, si l'état initial de l'automate est remplacé par p.*

Eisner attire l'attention sur le fait que l'algorithme de minimisation de Mohri (1997a) est limité aux machines à états finis définies sur des semi-anneaux qui acceptent la notion de *plus long préfixe commun*, comme le semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$ ou le semi-anneau des strings $(\Delta \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$ [5], parce que la préfixation pondérée proposée par Mohri nécessite de déterminer le plus long préfixe commun de la fonction suffixe F_p correspondant à chaque état p de la machine.

Or, Eisner constate que d'autres semi-anneaux pourraient tirer avantage d'un algorithme de minimisation. C'est le cas du semi-anneau réel $(\mathbb{R}, +, \times, 0, 1)$ et de son équivalent logarithmique.

Afin d'étendre la minimisation pondérée à ces semi-anneaux, Eisner propose de s'affranchir de la notion de *plus long préfixe commun*. L'algorithme d'Eisner ne diffère de celui de Mohri qu'au niveau de la phase de préfixation, et repose sur deux conditions qui caractérisent les semi-anneaux minimisables.

Semi-anneaux minimisables. Dans l'algorithme qu'il propose, Eisner ne s'intéresse pas à l'opération $\oplus$ du semi-anneau, parce qu'il considère des automates déterministes. De ce fait, il définit les semi-anneaux qui peuvent accepter la minimisation pondérées sur le monoïde $(\mathbb{K}, \otimes)$. Eisner démontre que les semi-anneaux minimisables présentent un $(\mathbb{K}, \otimes)$ qui autorise la *factorisation gourmande* et est tel que toute fonction suffixe F a *un résidu minimum*.

Factorisation gourmande. Etant donné une fonction suffixe F et un facteur gauche f' donné, on appelle H un *résidu* de F ssi $F = f' \otimes H$. Etant donné deux fonctions suffixes F et G, si $F = f' \otimes H$ et $G = g' \otimes H$, on dit que F et G ont un résidu commun et on écrit $F \simeq G$. Dans ces termes, si F et G sont des résidus de la même fonction non nulle et que $F \simeq G$, $(\mathbb{K}, \otimes)$ autorise ce qu'Eisner appelle la *factorisation gourmande* [6]. En somme, l'ordre dans lequel les facteurs gauches sont enlevés d'une fonction suffixe n'importe pas : on peut atteindre le même H canonique, que l'on divise d'abord par f ou par g.

Résidu minimum. L'idée à la base de la minimisation est de construire un automate M_{min} dont les états correspondent aux classes d'équivalence des états de M, et où chaque classe $[q] = \{q_1, \ldots, q_m\}$ a une fonction suffixe $F_{[q]}$. Par induction, on peut trouver au moins une fonction $F_{[q]}$ qui est un résidu commun de $F_{q_1}, \ldots, F_{q_m}$.

Si M a une transition $p \xrightarrow{a/k} q$, M_{min} a besoin d'une transition $[p] \xrightarrow{a/k'} [q]$ où k est tel que $a^{-1}F_{[p]} = k' \otimes F_{[q]}$. La difficulté principale est d'assurer que k' existe effectivement. $F_{[q]}$ doit donc être choisi de telle sorte qu'il constitue non seulement un résidu commun à $F_{q_1}, \ldots, F_{q_m}$, mais également à $a^{-1}F_{[p]}$.

On appelle *résidu minimum* de $F \neq 0$ le résidu commun à tous les résidus de F. Si $(\mathbb{K}, \otimes)$ est tel que tout F a un résidu minimum, alors F a le même résidu minimum que

[5] Dans ce semi-anneau, Δ est un alphabet et correspond à l'alphabet de sortie d'un transducteur.
[6] *Greedy factorization* dans l'article.

Require: Un WFSA $A_K = (\Sigma, Q, i, F, E, \rho)$ sur le semi-anneau $\mathbb{K}$
Ensure: Le WFSA $A_{K_P} = (\Sigma, Q, i, F, E', \rho')$ dont les poids ont été poussés
1: $V_{min} \leftarrow$ `BESTFROMFINAL`(A_K)
2: $E' \leftarrow \emptyset$
3: $\rho'[p] \leftarrow 0$ **for** $p \leftarrow 0$ **to** $|Q| - 1$
4: **for each** $p \in Q$ **do**
5: **for each** $e \in E[p]$ **do**
6: $w' \leftarrow V_{min}[e.p]^{-1}\, e.w + V_{min}[e.q]$
7: $E' \leftarrow E' \cup \{(e.p, e.a, w', e.q)\}$
8: **end for**
9: **end for**
10: $A_{K_P} \leftarrow (\Sigma, Q, i, F, E', \rho')$
11: **return** A_{K_P}

Pseudocode 22: `W-PREFIXATION`

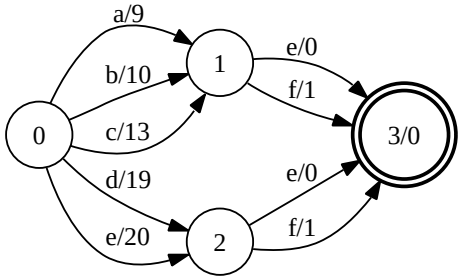


FIG. 4.6: Automate pondéré préfixé A_{K_3}, équivalent à A_{K_2}

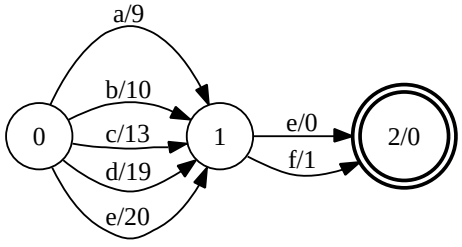


FIG. 4.7: Automate pondéré minimal A_{K_4}, équivalent à A_{K_3}

tout $H \simeq F$.

Dans un tel $(\mathbb{K}, \otimes)$, la fonction $F_{[q]}$ peut être choisie indépendamment. Par exemple, $F_{[q]} = F_{q_1}, \ldots, F_{q_m}$ peut être le résidu minimum de F_{q_1}.

De ce fait, pour la transition $[p] \xrightarrow{a/k'} [q]$, étant donné que $a^{-1}F_{[p]} \simeq a^{-1}F_q \simeq F_q \simeq F_{q_1}$, $F_{[q]}$ est le résidu minimum de $a^{-1}F_{[p]}$, ce qui assure que k' est défini pour $a^{-1}F_{[p]} = k' \otimes F_{[q]}$.

Notation. Chez Eisner, $k \backslash m$ note le quotient gauche k de m. Eisner emploie cette notation plutôt que $k^{-1} \otimes m$, parce que k^{-1} fair référence à un véritable élément de $\mathbb{K}$, alors que $k \setminus m$ ne doit pas forcément exister. Un exemple : $ww \setminus wwzzz = zzz$, mais $wwy \setminus wwzzz$ n'existe pas.

Nouvelle fonction. L'algorithme de pondération d'Eisner repose sur la définition, par l'utilisateur, d'une fonction $\lambda(F)$, capable de déterminer un facteur gauche pour toute fonction F : étant donné $F = (\Sigma^* \to \mathbb{K})$, $\lambda(F) \to \mathbb{K}$. La fonction λ doit respecter les propriétés suivantes :

1. *Shift* : $\lambda(k \otimes F) = k \otimes \lambda(F)$.
2. *Quotient* : $\lambda(F) \setminus \lambda(a^{-1}F)$ existe dans $\mathbb{K}$ pour tout $a \in \Sigma$.
3. *Quotient final* : $\lambda(F) \setminus \lambda(\epsilon)$ existe dans $\mathbb{K}$.

Algorithme. Etant donné un automate pondéré $A_K = (\Sigma, Q, i, F, d, w, \rho)$, la préfixation pondérée proposée par Eisner se divise en deux étapes. La première étape permet de calculer le facteur gauche $\lambda(F_p)$ de chaque état $p \in Q$ selon la définition suivante :

$$\lambda(F) = \begin{cases} F(\min supp(F)) \in \mathbb{K} & \text{si } F \neq \underline{0} \\ \underline{0} & \text{si } F = \underline{0} \end{cases}$$

où $supp(F)$ correspond à l'ensemble des strings d'entrée auxquelles F assigne un poids différent de $\underline{0}$. Les strings de $supp(F_p)$ sont classées d'abord en fonction de la longueur, ensuite en fonction de l'ordre alphabétique, de sorte que $\epsilon < bb < aab < aba < abb$.

Sur cette base, $\min supp(F_p)$ est la string la plus courte de $supp(F_p)$, et le facteur $\lambda(F_p)$ est le poids de cette string. $\lambda(F_p)$ est obtenu facilement, étant donné que :

1. La string la plus courte de $supp(F_p)$ ne présente pas de cycles.
2. S'il existe une transition $p \xrightarrow{a/k} q$, $\lambda(F_p)$ dépend de la définition de $\lambda(F_q)$:

$$\lambda(F_p) = k \otimes \lambda(F_q)$$

Ceci étant posé, l'algorithme permettant de calculer les facteurs $\lambda(F_p)$ est très simple. Il est illustré en Pseudocode 23. Chaque état et chaque transition y sont considérés une seule fois, au cours d'une recherche en largeur d'abord, initialisée à partir des états finaux. L'algorithme utilise pour ce faire une file de type FIFO [7]. Les variables $len(p)$ et

[7] FIFO : *First In, First Out.*

Require: Un WFSA $A_K = (\Sigma, Q, i, F, E, \rho)$ sur le semi-anneau $\mathbb{K}$
Ensure: Pour chaque $p \in Q$, $\lambda(F_p)$ est calculé
1: $C \leftarrow \emptyset$
2: **for each** $p \in Q$ **do**
3: **if** $p \in F$ **then**
4: $len(p) \leftarrow 0$
5: $\lambda(F_p) \leftarrow \rho(p)$
6: ENQUEUE_FIFO(C, p)
7: **else**
8: $len(p) \leftarrow$ NULL
9: $\lambda(F_p) \leftarrow 0$
10: **end if**
11: **end for**
12: **while** $C \neq \emptyset$ **do**
13: $q \leftarrow$ DEQUEUE_FIFO(C)
14: **for each** $e \in E_r[q] : e.w \neq 0$ **do**
15: **if** $len(e.p) =$ NULL **then**
16: ENQUEUE_FIFO$(C, e.p)$
17: **end if**
18: **if** $len(e.p) =$ NULL **or** $[len(e.p) = len(q) + 1$ **and** $e.a < first(e.p)]$ **then**
19: $first(e.p) \leftarrow e.a$
20: $len(e.p) \leftarrow len(q) + 1$
21: $\lambda(F_{e.p}) \leftarrow e.w \otimes \lambda(F_q)$
22: **end if**
23: **end for**
24: **end while**

Pseudocode 23: COMPUTE-L

$first(p)$ mémorisent respectivement la longueur et la première lettre de $\min supp(F_p)$.

Lorsque tous les facteurs λ sont calculés, la deuxième étape de l'algorithme remplace le poids k d'une transition $p \xrightarrow{a/k} q$ par k', calculé comme suit :

$$\begin{aligned} k' &= \lambda(F_p) \setminus \lambda(a^{-1}F_p) \\ &= \lambda(F_p) \setminus \lambda(k \otimes F_q) \\ &= \lambda(F_p) \setminus k \otimes \lambda(F_q) \end{aligned}$$

bien défini par la propriété de *Shift*. Au cours de cette seconde étape, l'algorithme met également à jour les poids émis par ρ pour chaque état final p :

$$\rho(p) = \lambda(F_p) \setminus \rho(p)$$

bien défini par la propriété de *Quotient final*. Cette mise à jour des transitions a été initialement prévue pour les semi-anneaux à division, comme le semi-anneau réel $(\mathbb{R}, +, \times, 0, 1)$.

Elle peut cependant être appliquée aux semi-anneaux sans division, pour autant qu'ils soient étendus par l'ajout des inverses pour $\otimes$, de manière à assurer la définition de $\lambda(F_p)\backslash k \otimes \lambda(F_q)$. Par exemple, le semi-anneau tropical peut être étendu par l'ajout des nombres réels négatifs : $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$. De même, le semi-anneau des strings $(\Delta \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$ peut être étendu en augmentant $\Delta = \{a, b, c, \ldots\}$ des lettres inversées $\{a^{-1}, b^{-1}, c^{-1}, \ldots\}$, ce qui permet d'appliquer l'algorithme d'Eisner à des transducteurs.

Complexité. Contrairement à l'algorithme de Mohri (1997a) qui n'est plus linéaire lorsque la machine à minimiser contient des cycles, ce nouvel algorithme reste constamment linéaire ($O(|Q| + |E|)$). Eisner (2003) note cependant que l'algorithme de Mohri est légèrement plus rapide lorsqu'il est linéaire.

Comparaison des deux méthodes. La Figure 4.8 illustre la différence fondamentale qui distingue l'algorithme d'Eisner (2003) de celui de Mohri (1997a), lorsque les deux algorithmes sont appliqués à un transducteur.

Dans les deux cas, on constate que la préfixation met au jour les classes d'équivalence $\{1, 3\}$ et $\{2, 4\}$. Les transitions obtenues par les deux algorithmes sont cependant totalement différentes. Le résultat de Mohri (b), obtenu par détection du *plus long préfixe commun*, est bien sûr le résultat attendu.

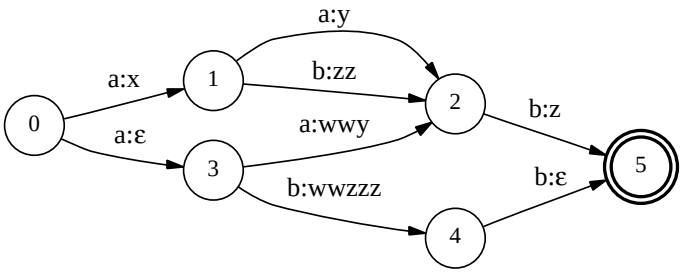
Dans le résultat obtenu par Eisner, les symboles en majuscules $(X, Y, \ldots)$ font référence aux symboles inverses $(x^{-1}, y^{-1}, \ldots)$ ajoutés à Δ. La présence de ces symboles sur les transitions est le résultat du calcul de $k' = \lambda(F_p) \setminus k \otimes \lambda(F_q)$. Par exemple, la transition $1 \xrightarrow{b:zz} 2$ aboutit à $1 \xrightarrow{b:z^{-1}y^{-1}zzz} 2$ comme suit :

$$\begin{array}{rcllll} k' & = & \lambda(2) & \backslash & zz & \otimes & \lambda(3) \\ & = & yz & \backslash & zz & \otimes & z \\ & = & (yz)^{-1} & \otimes & zz & \otimes & z \\ & = & z^{-1}y^{-1} & \otimes & zz & \otimes & z \\ & = & z^{-1}y^{-1}zzz & & & & \end{array}$$

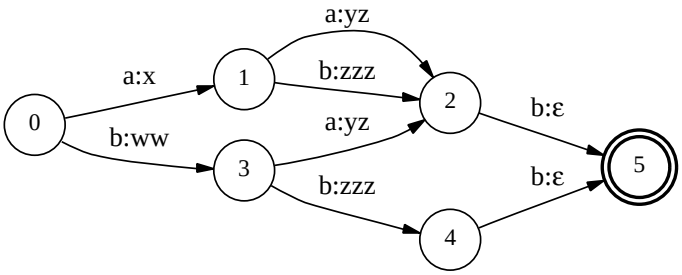
Ce résultat peut cependant être aisément converti en celui obtenu par Mohri. Le mieux semble de réaliser cette conversion *pendant* la minimisation classique, lorsque les classes d'équivalence ont été détectées (cf. Figure 4.8, (d)). Le principe est le suivant : pour une classe d'équivalence $\{p_1, p_2, \ldots, p_j\}$ qui correspondra à l'état p dans la machine minimisée, il faut déterminer le plus long suffixe commun (LCS [8]) des facteurs gauches de chaque état de la classe :

$$\texttt{LCS}(p) = \texttt{LCS}\{\lambda(p_i) : i = 1, 2, \ldots, j\}$$

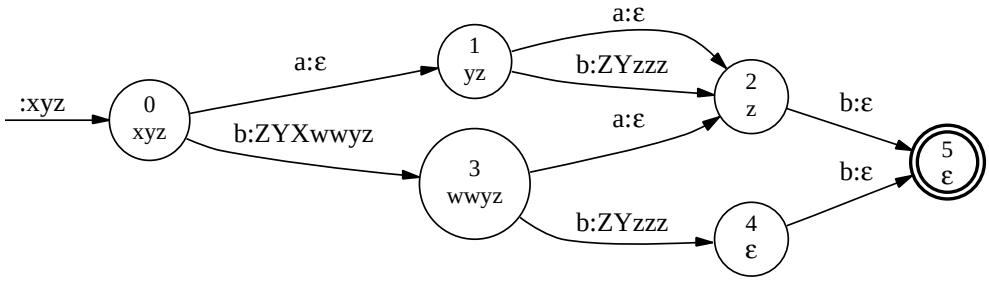
[8] LCS : *Longest Common Suffix*, plus long suffixe commun.



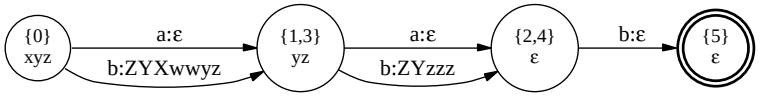
(a) avant préfixation



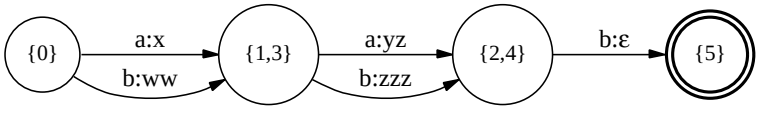
(b) préfixation de Mohri (1997a)



(c) préfixation d'Eisner (2003)



(d) minimisation – plus longs suffixes communs (Eisner 2003)



(e) machine minimisée

FIG. 4.8: Comparaison des algorithmes de préfixation pondérée. Figures (a,b,c) reprises de Eisner (2003)

Dans notre exemple,

$$\begin{aligned}
\texttt{LCS}(\{0\}) &= xyz \\
\texttt{LCS}(\{1,3\}) &= yz \\
\texttt{LCS}(\{2,4\}) &= \epsilon \\
\texttt{LCS}(\{5\}) &= \epsilon
\end{aligned}$$

Dans le machine minimisée, une transition $p \xrightarrow{a/k} q$ sera ensuite réécrite $p \xrightarrow{a/k'} q$ avec :

$$k' = \texttt{LCS}(p) \otimes k \otimes \texttt{LCS}(q)^{-1}$$

Par exemple, pour $\{0\} \xrightarrow{a/\epsilon} \{1,3\}$:

$$\begin{aligned}
k' &= \texttt{LCS}(\{0\}) \otimes \epsilon \otimes \texttt{LCS}(\{1,3\})^{-1} \\
&= (xyz) \otimes \epsilon \otimes (yz)^{-1} \\
&= x
\end{aligned}$$

et pour $\{0\} \xrightarrow{b/(xyz)^{-1}wwyz} \{1,3\}$:

$$\begin{aligned}
k' &= \texttt{LCS}(\{0\}) \otimes (xyz)^{-1}wwyz \otimes \texttt{LCS}(\{1,3\})^{-1} \\
&= (xyz) \otimes (xyz)^{-1}wwyz \otimes (yz)^{-1} \\
&= ww
\end{aligned}$$

Après minimisation, les résultats obtenus par les deux méthodes de minimisation sont donc identiques (cf. Figure 4.8, (e)).

4.7 Espérance-Maximisation

La construction de machines pondérées s'inscrit classiquement dans deux traditions. D'une part, la tradition algébrique, qui recourt à des experts du domaine étudié afin de modéliser manuellement des expressions régulières qui peuvent être pondérées. Cette modélisation donne lieu à des langages complexes qui nécessitent la définition de nombreux opérateurs, dérivés des opérateurs rationnels standard (Karttunen *et al.* 1996). D'autre part, la tradition statistique évoquée en début de chapitre (Pereira & Sproat 1994, Pereira & Riley 1997, Mohri *et al.* 1996), qui préfère déterminer la probabilité d'un événement à partir de données, et génère des modèles de langue fort simples. Les modèles que nous proposons dans le cadre de cette thèse (cf. Chapitres 10, 15 et 16) s'inscrivent dans ces deux traditions.

Il faut néanmoins noter que Eisner (2001, 2002a) a proposé une méthode permettant de combiner les deux traditions. L'idée est d'estimer automatiquement, à l'aide d'un algorithme d'Espérance-Maximisation (Dempster *et al.* 1977), les paramètres intervenant dans des règles construites manuellement par des experts. Pour ce faire, Eisner propose un nouveau type de semi-anneau, qui permet de calculer l'Espérance-Maximisation au travers de machines à états finis. Ceci s'éloignant significativement des méthodes que nous avons appliquées, nous laissons le lecteur intéressé consulter les articles mentionnés.

4.8 Synthèse

Sur la base de l'équivalence établie entre machines pondérées et séries rationnelles, la notion de *distance la plus courte à partir d'une source unique* définie sur les semi-anneaux k-clos a permis la modélisation d'algorithmes efficaces d'optimisation et de recherche des n meilleurs chemins d'un graphe pondéré. Les machines pondérées complètent dès lors les automates et transducteurs classiques en leur ajoutant la possibilité de gérer un certain degré d'incertitude, nécessaire à certaines applications en traitement des langues, comme celles que nous présentons dans les Parties II et III.

Chapitre 5

Expressions régulières et règles de réécriture

5.1 Introduction

Les automates et les transducteurs, pondérés et non pondérés, sont indéniablement des outils puissants. Cependant, dès que le langage à représenter devient un peu complexe, l'automate correspondant est difficile à construire « manuellement ». La tâche est encore plus ardue lorsqu'il s'agit de construire le transducteur équivalent à une relation.

Il est donc nécessaire de posséder un mode de description compact et aisé qui permette de définir langages et relations. Les expressions régulières et les règles de réécriture sont les modes de description à partir desquels il est possible de construire les machines à états finis équivalentes.

5.2 Les expressions régulières

5.2.1 Définition

Une expression régulière est une description algébrique permettant de représenter de manière concise un langage régulier.

Les algèbres de tout type reposent sur des expressions élémentaires, traditionnellement des constantes et/ou des variables. Les algèbres permettent ensuite de construire d'autres expressions en appliquant un ensemble d'opérateurs à ces expressions élémentaires ainsi qu'à des expressions construites précédemment. Traditionnellement, une méthode permettant de grouper les opérateurs, comme les parenthèses, est nécessaire également.

L'algèbre sur lequel les expressions régulières sont fondées respecte ce formalisme décrit : il utilise des constantes et des variables qui représentent des langages, et les combine à l'aide des opérateurs *concaténation*, *union* et *étoile de Kleene*.

Définition 5.2.1. *Posons Σ, un alphabet. Les* expressions régulières *sur Σ et les langages qu'elles représentent sont définis récursivement comme suit :*

1. *La variable L note un langage.*
2. *$\emptyset$ est une expression régulière qui représente le langage $\emptyset$: $L(\emptyset) = \emptyset$.*
3. *ϵ est une expression régulière qui représente le langage $\{\epsilon\}$: $L(\epsilon) = \{\epsilon\}$.*
4. *Pour tout symbole a appartenant à Σ, a est une expression régulière qui représente le langage $\{a\}$: $L(a) = \{a\}$.*
5. *Si e et f sont des expressions régulières qui représentent respectivement les langages réguliers $L(e)$ et $L(f)$, alors*
   - *union : $e \mid f$ est une expression régulière qui représente le langage $L(e|f)$ tel que $L(e|f) = L(e) \cup L(f)$,*
   - *concaténation : $e \cdot f$ est une expression régulière qui représente le langage $L(e \cdot f)$ tel que $L(e \cdot f) = L(e) \cdot L(f)$ et*
   - *étoile de Kleene : e^* est une expression régulière qui représente le langage $L(e^*)$ tel que $L(e^*) = (L(e))^*$.*
6. *Si e est une expression régulière qui représente le langage régulier $L(e)$, l'expression parenthésée (e) représente le même langage régulier : $L((e)) = L(e)$.*

Note 5.2.1.

- Le point qui marque la concaténation n'est jamais utilisé dans l'écriture des expressions régulières. Ainsi, $e \cdot f$ s'écrit ef.
- Dans l'écriture d'une expression régulière, on peut supprimer de nombreuses parenthèses, pour autant que l'on s'accorde sur la précédence suivante définie sur les opérateurs :

  étoile de Kleene > concaténation > union (5.2.1.1)

  Ceci signifie que :
  1. L'étoile de Kleene a la plus haute précédence. Elle ne s'applique donc qu'à l'opérande située directement à sa gauche, qui est la plus petite expression régulière correctement formée. ainsi, ab^* équivaut à $a(b^*)$ et non à $(ab)^*$
  2. Ensuite vient la concaténation. Ainsi, toutes les expressions directement juxtaposées sont regroupées dès que l'étoile de Kleene a été appliquée. La concaténation est associative ; l'ordre du regroupement importe donc peu : $(ab)c = a(bc)$.
  3. Enfin, l'union peut s'appliquer. L'union est aussi associative : $(a \mid b) \mid c = a \mid (b \mid c)$. Cependant, nous supposons un regroupement suivant l'ordre de la lecture, de gauche à droite.

  Cette précédence assure par exemple que $((a(b^*)) \mid c) = ab^* \mid c$. Les parenthèses sont dès lors plutôt employées afin de modifier la précédence établie. Par exemple, si l'on désire réaliser l'union avant la concaténation, on écrira $a(b^* \mid c)$ au lieu de $ab^* \mid c$.

5.2.2 De l'expression régulière à l'automate

Convention 5.2.1. Pour lever toute ambiguïté dans les figures qui illustrent cette section, l'état initial est représenté par une flèche entrante, et l'état final est en gras et possède une flèche sortante. la Figure 5.1 explicite ces conventions.

FIG. 5.1: Conventions

Bien que les expressions régulières et les automates à états finis décrivent les langages de manières fondamentalement différentes, Kleene a montré l'équivalence de ces deux notations (cf. Section 2.5). Ceci implique que :

1. Tout langage défini par une expression régulière peut également être défini par un automate.
2. Tout langage défini par un automate peut également être défini par une expression régulière.

Dans le contexte du travail présenté dans ce document, nous nous intéressons exclusivement au premier cas, c'est-à-dire à la possibilité de construire l'automate équivalent à une expression régulière donnée. Le lecteur intéressé par l'opération inverse se référera utilement à (Aho *et al.* 1974, Hopcroft *et al.* 1979).

McNaughton & Yamada (1960) on démontré qu'il est possible de construire linéairement en terme de temps et d'espace un ϵ-NFA qui représente une expression régulière.

Les conventions qu'ils ont posées sont que tous les automates construits pour la représentation d'une expression régulière sont des ϵ-NFAs :

1. pourvus d'un seul état final
2. sans transition entrant dans l'état initial
3. sans transition quittant l'état final

Pourquoi ? Leur constat est qu'un automate qui représenterait une expression régulière, mais ne respecterait pas ces conventions ne pourrait intervenir dans la construction d'un autre automate représentant une expression régulière plus complexe. En voici un exemple, illustré par la Figure 5.2 : l'expression « $a^* \mid b$ » représente le langage « soit zéro, un ou plusieurs a, soit un b ». Si l'on part des automates 1 et 2, on obtient l'automate 3, qui représente le langage « zéro, un ou plusieurs a *suivis* d'un b ». L'erreur est due à l'automate 1, pourvu d'un seul état à la fois initial et final, qui enfreint deux des conventions posées, puisque la transition a sort de l'état final et entre dans l'état initial.

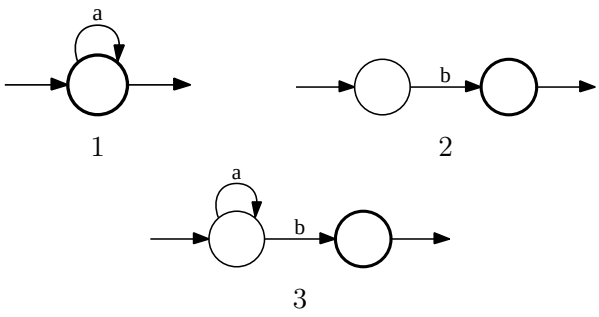


FIG. 5.2: Conversion « expression régulière → automate » erronée

Ainsi, l'idée de l'algorithme est de construire l'automate correspondant à l'expression régulière *de proche en proche* à partir d'expressions de base. Nous commençons par montrer comment construire les automates correspondant aux expressions de base : $\emptyset$, ϵ et a. Nous montrons ensuite comment combiner ces automates de base dans des automates plus grands qui acceptent l'union, la concaténation et l'étoile de Kleene.

Base. La Figure 5.3 présente les automates correspondant aux expressions de base.

1. L'automate 1 représente l'expression $\emptyset$: on constate qu'effectivement le langage modélisé est $\emptyset$, puisqu'il n'y a pas de chemin de l'état initial à l'état final.
2. L'automate 2 représente l'expression ϵ : le langage modélisé est $\{\epsilon\}$, car le seul chemin menant de l'état initial à l'état final est étiqueté ϵ.
3. Enfin, l'automate 3 représente l'expression a, et le langage modélisé est $\{a\}$ puisque le seul chemin menant de l'état initial à l'état final est étiqueté a.

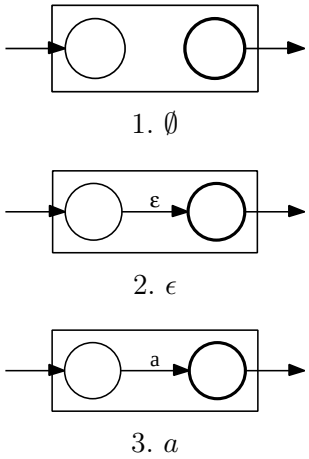
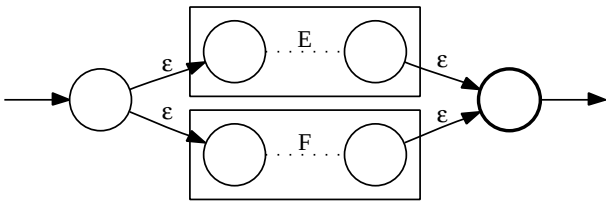
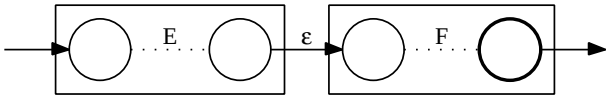


FIG. 5.3: Expressions régulières de base

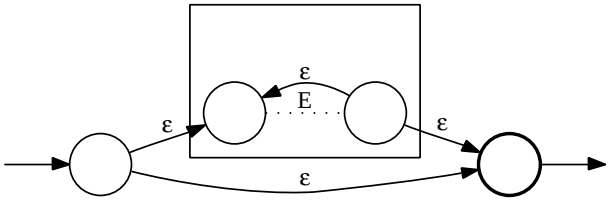
Induction. La Figure 5.4 illustre les principes mis en œuvre pour représenter sous la forme d'un automate l'application d'un opérateur à une expression régulière. Dans cette figure, E et F sont les automates précédemment obtenus pour les expressions régulières e et f, et nous supposons que ces automates sont des ϵ-NFAs qui ont été construits selon les conventions posées. Trois cas de figure sont à prendre en compte :



1. union



2. concaténation



3. étoile de Kleene

FIG. 5.4: Expressions régulières combinées par les opérateurs

1. L'expression est « $e \mid f$ » : union (automate 1). Les automates E et F sont réunis dans un nouvel automate G. Dans G, un nouvel état initial est créé, et possède des transitions ϵ vers les anciens états initiaux de E et F. En outre, dans G, un nouvel état final est créé, et les anciens états finaux de E et F possèdent chacun une transition ϵ vers ce nouvel état final. Clairement, $L(G)$ est bien équivalent à $L(E) \cup L(F)$, puisque l'on peut parcourir E et F dans G, mais un seul à la fois.

2. L'expression est « ef » : concaténation (automate 2). Les automates E et F sont réunis dans un nouvel automate G. On constate que l'état initial de G est l'état initial de E, et que l'état final de G est l'état final de F. En outre, l'ancien état final de E est désormais lié à l'ancien état initial de F *via* une transition ϵ. En somme, tout chemin valide dans G doit d'abord traverser complètement E avant de traverser complètement F. Donc, $L(G)$ correspond bien à $L(E) \cdot L(F)$.

3. L'expression est « e^* » : étoile de Kleene (automate 3). A partir de l'état initial de l'automate G construit sur E, deux solutions sont possibles :

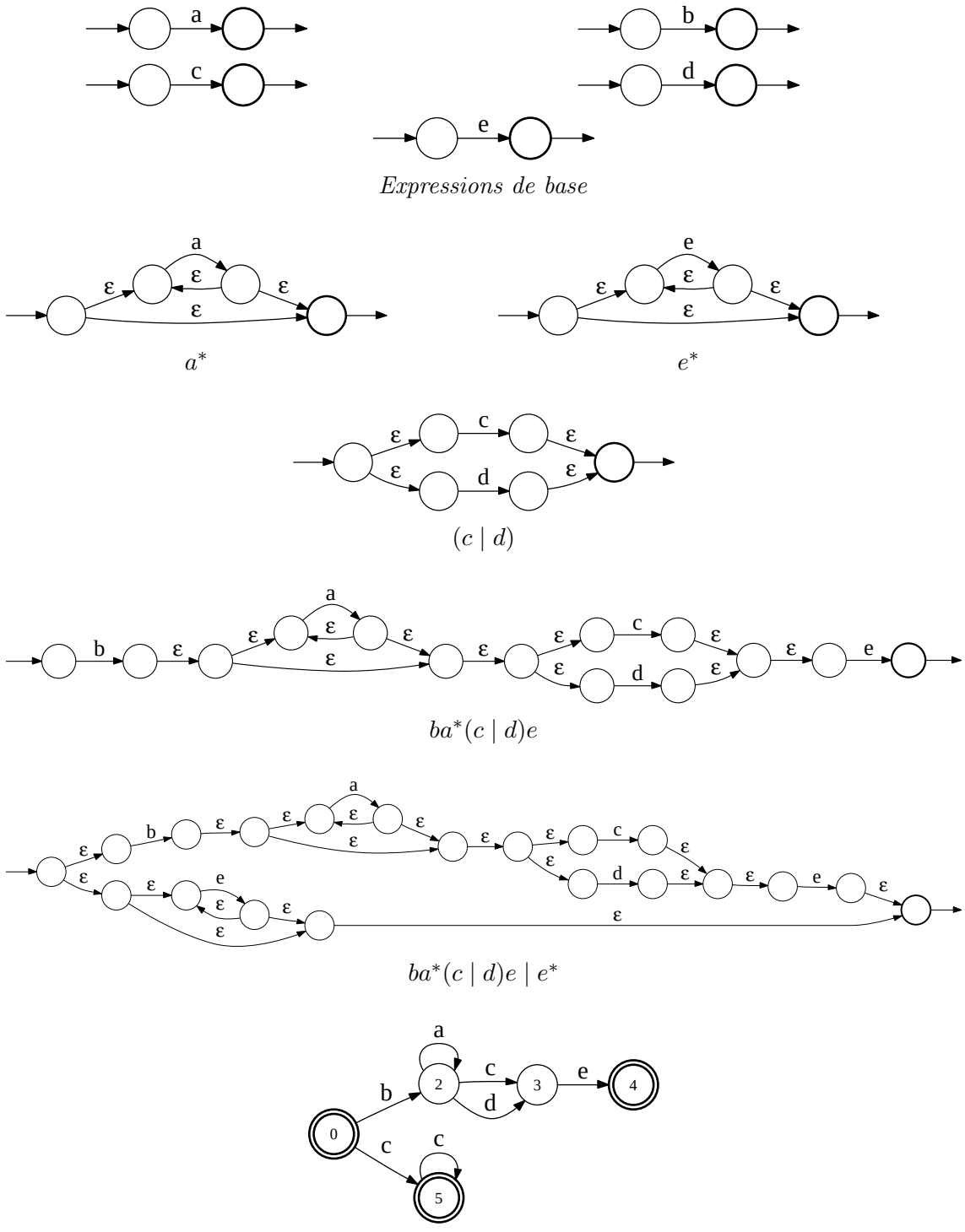
(a) Soit rejoindre directement le nouvel état final *via* une transition ϵ. Le langage représenté est ici $\{\epsilon\}$, qui appartient à $L(E)^*$.

(b) Soit rejoindre l'ancien état initial de E, et parcourir *au moins une fois* E, avant de rejoindre le nouvel état final à partir de l'ancien état final de E. Le langage représenté correspond donc à $L(E)^*$, sauf $\{\epsilon\}$ pour autant que $\{\epsilon\}$ ne fasse pas partie de $L(E)$. Cependant, $\{\epsilon\}$ est de toute façon couvert par la solution précédente.

On constate en outre que les trois automates respectent les conditions posées, puisqu'ils ne possèdent qu'un seul état final sans transition sortante, et que l'état initial ne possède pas de transition entrante.

Simplification. Lorsque l'ϵ-NFA correspondant à la totalité de l'expression régulière a été construit, il est évidemment conseillé de lui appliquer les algorithmes *suppression-ϵ*, *déterminisation* et *minimisation*.

Un exemple complet. La Figure 5.5 détaille le déroulement de l'algorithme pour la construction de l'automate équivalent à l'expression régulière « $ba^*(c \mid d)e \mid e^*$ ».



Expressions de base

a^*

e^*

$(c \mid d)$

$ba^*(c \mid d)e$

$ba^*(c \mid d)e \mid e^*$

L'automate après simplification

FIG. 5.5: « $ba^*(c \mid d)e \mid e^*$ » : de l'expression à l'automate

5.3 Les règles de réécriture

5.3.1 Introduction

Les règles de réécriture prennent la forme générale suivante :

$$\phi \Rightarrow \psi :: \lambda \_ \rho \tag{5.3.1.1}$$

qui exprime que ϕ se réécrit ψ dans le contexte de λ et de ρ. Des quatre parties de la règle, ϕ est la *cible*, ψ est la *réécriture*, λ est le *contexte gauche* et ρ, le *contexte droit*. En outre, $\phi \Rightarrow \psi$ est appelé le *remplacement*. Les quatre parties d'une règle sont généralement considérées comme des expressions régulières, bien que certains préfèrent limiter la cible et le remplacement à de simples strings, voire à des symboles uniques.

Développées dans le contexte de la phonologie générative, les règles de réécriture sont maintenant largement utilisées dans de nombreux domaines, dont celui du traitement automatique du langage (*e.g.*, pré-traitement ou analyse morpho-syntaxique).

La description complète d'un domaine d'application nécessite généralement de nombreuses règles de réécriture. Or, dans la tradition issue du formalisme chomskyen (Chomsky & Halle 1968), trois contraintes sont exprimées sur les ensembles de règles de réécriture :

1. Les règles sont ordonnées de la plus spécifique à la plus générale, de sorte qu'une règle ne puisse s'appliquer que si aucune règle, plus spécifique et plus appropriée, n'a été rencontrée précédemment.
2. Une règle ne peut réécrire sa propre sortie, mais cette sortie peut servir de *contexte* à la réapplication de cette même règle. Cette contrainte interdit donc qu'une règle comme

   $$\epsilon \Rightarrow ab :: a \_ b \tag{5.3.1.2}$$

   qui insère ab entre a et b, puisse réécrire sa sortie elle-même. Ceci reviendrait à projeter ab sur le langage *hors-contexte* $\{a^n b^n : n \geq 1\}$.
3. Tout contexte d'une règle donnée (λ, ρ), s'il est la cible d'une autre règle, sera également réécrit.

En somme, toutes les règles applicables à une entrée donnée lui seront appliquées, simultanément ou séquentiellement selon la méthode de représentation choisie, mais certainement pas *récursivement*. Les règles de réécriture ne modélisent donc pas de langages hors-contexte.

L'efficacité du système dépendra évidemment énormément de la méthode de représentation choisie. En effet, le respect des contraintes exprimées sur les règles de réécriture peut conduire à la modélisation d'un système naïf qui testerait séquentiellement l'applicabilité de chaque règle sur l'entrée en cours d'analyse. Un tel système, pour un ensemble de règles R et une string d'entrée u, serait de complexité $O(|R| \times |u|)$.

Johnson (1972) a été le premier à démontrer que les règles de réécriture [1] modélisent des langages réguliers, et sont dès lors représentables par des transducteurs. En effet,

[1] Dans son cas, les règles étaient exclusivement celles d'un système phonologique.

- chaque partie d'une règle, qu'elle soit considérée comme une expression régulière, une simple string ou un unique symbole, peut être représentée de manière compacte sous la forme d'un automate à états finis,
- la règle en elle-même est un exemple de transduction entre deux langages : le langage d'entrée, $\lambda\,\phi\,\rho$, et le langage de sortie, $\lambda\,\psi\,\rho$,
- le principe de composition en lui-même modélise parfaitement les trois contraintes exprimées sur les ensembles de règles de réécriture : un seul transducteur, résultant de la composition ordonnée de l'ensemble de règles, peut représenter et appliquer simultanément l'ensemble des règles applicables à une entrée donnée.

L'avantage indéniable, dans le cas de l'utilisation de machines à états finis pour modéliser les règles de réécriture, est l'obtention d'un système dont le temps de calcul, pour un ensemble de règles R et une string d'entrée u, est de complexité $O(|u|)$. Ceci est dû, bien sûr, à l'arsenal algorithmique défini sur les machines qui permet d'obtenir des machines déterministes.

5.3.2 Types de règles : variations sur un même thème

Les règles de réécriture autorisent certaines variantes très utiles, combinaisons des critères suivants : contexte, mode d'application, optionalité et pondération.

Absence de contexte. Si l'un ou l'autre contexte est vide, il n'est pas indiqué, et si les deux sont vides, la règle se réduit à :

$$\phi \Rightarrow \psi \tag{5.3.2.1}$$

Mode d'application. Les phonologues ont proposé trois modes d'application possibles pour une règle donnée. (1) Soit une application *de gauche à droite*, où chaque application est réalisée sur la *sortie* de l'application précédente. (2) Soit la stratégie inverse, l'application *de droite à gauche*, où chaque application est également réalisée sur la *sortie* de l'application précédente. (3) Soit une application *simultanée* de la règle, où chaque possibilité d'application est identifiée dans la string d'*origine* et non dans la sortie de l'application précédente. Si l'on prend une règle telle que :

$$a \rightarrow b :: ab\ \_\ ba$$

et qu'elle est appliquée à la string :

abababababa

les différents modes d'application donneront :

(1) de gauche à droite : $ab\underline{b}ab\underline{b}baba$
(2) de droite à gauche : $ababb\underline{b}ab\underline{b}ba$
(3) simultanément : $abb\underline{b}b\underline{b}b\underline{b}ba$

Si les trois stratégies ont indéniablement un intérêt, il est cependant nécessaire de les distinguer de manière non ambiguë dans l'*expression* de la règle :

1. Application de gauche à droite :

$$\phi \rightarrow \psi :: \lambda \_ \rho \tag{5.3.2.2}$$

2. Application de droite à gauche :

$$\phi \leftarrow \psi :: \lambda \_ \rho \tag{5.3.2.3}$$

3. Application simultanée :

$$\phi \leftrightarrow \psi :: \lambda \_ \rho \tag{5.3.2.4}$$

Optionalité. Une règle peut être facultative. Dans ce cas, son application sur une string d'entrée donnera au moins deux résultats selon le mode d'application choisi : la string elle-même, sans aucune modification, et la string dans laquelle tous les remplacements autorisés par le mode d'application choisi auront été réalisés. Les règles optionnelles se différencient par l'ajout d'un point d'interrogation sur la flèche du mode d'application :

$$\begin{aligned} &\phi \xrightarrow{?} \psi :: \lambda \_ \rho \\ &\phi \xleftarrow{?} \psi :: \lambda \_ \rho \\ &\phi \xleftrightarrow{?} \psi :: \lambda \_ \rho \end{aligned} \tag{5.3.2.5}$$

La représentation de la règle au moyen d'un transducteur de type séquentiel permettra de conserver une entrée déterministe, et donc une complexité $O(|u|)$ pour une string d'entrée u (cf. Section 3.4).

Pondération. Quel que soit le mode d'application et que la règle soit optionnelle ou obligatoire, il peut être intéressant qu'un remplacement soit accompagné d'un poids. Ce poids permettra, dans un système pondéré, de tenir compte d'une certaine incertitude et de choisir, à terme, le meilleur chemin dans un graphe de possibilités :

$$\begin{aligned} &\phi \xrightarrow{?} \psi_1 :: \lambda \_ \rho \,/\, w_1 \\ &\phi \rightarrow \psi_2 :: \lambda \_ \rho \,/\, w_2 \end{aligned} \tag{5.3.2.6}$$

où w_1 et w_2 sont les poids associés aux règles de réécriture.

5.3.3 Algorithmes

Les principaux algorithmes permettant de compiler des règles de réécriture sous la forme de transducteurs sont présentés dans (Kaplan & Kay 1994, Karttunen 1995, Mohri & Sproat 1996).

Quel que soit l'algorithme, le principe général mis en œuvre est le même. Chaque règle est d'abord compilée séparément. Les règles sont ensuite composées ensemble dans l'ordre dans lequel elles sont exprimées. La compilation d'une règle varie fortement d'un algorithme à l'autre, mais se base toujours sur les principes suivants :

1. Chaque partie d'une règle est considérée comme une expression régulière définie sur l'alphabet des règles, et compilée sous la forme d'un automate. Les quatre automates sont la base des étapes suivantes.
2. Le remplacement d'une règle correspond au produit cartésien $\phi \times \psi$, qui peut être obtenu en composant un transducteur $T_{\phi:\epsilon}$, qui projette l'expression ϕ sur ϵ, avec un transducteur $T_{\epsilon:\psi}$, qui projette ϵ sur l'expression ψ :

$$\phi \times \psi = T_{\phi:\epsilon} \circ T_{\epsilon:\psi} \tag{5.3.3.1}$$

   La Figure 5.6 donne un exemple de produit cartésien pour $\phi = AA$ et $\psi = B$.
3. Pour chaque règle, des transducteurs intermédiaires sont nécessaires. Leur objectif est d'identifier de manière non ambiguë les contextes gauche et droit de la règle, de sorte que le remplacement ne soit réalisé que dans le contexte valide. Dans ces transducteurs, les contextes sont identifiés à l'aide de marqueurs qui seront finalement supprimés. Les transducteurs intermédiaires travaillent tous sur la totalité de l'alphabet des règles, de manière à accepter toute string définie sur l'alphabet. Ceci a pour conséquence que le processus dans sa globalité dépend de la taille de l'alphabet et peut être fort coûteux. Néanmoins, la complexité d'insertion des marqueurs diffère considérablement selon l'algorithme.
4. Tous ces transducteurs intermédiaires sont composés ensemble, de manière à créer un seul transducteur qui modélise la règle et dans lequel les marqueurs ont disparu.

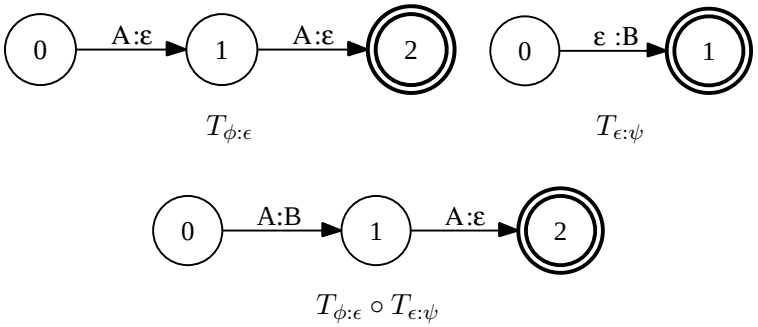


FIG. 5.6: Construction du remplacement $AA \rightarrow B$

Kaplan & Kay. Dans cet algorithme qui est le premier du genre, l'insertion des marqueurs dans les divers transducteurs intermédiaires implique un grand nombre d'opérations coûteuses, telles que l'intersection, la soustraction, le complément et la déterminisation. Cet algorithme a en outre besoin d'un transducteur *dédié* à la suppression des marqueurs utilisés par les transducteurs précédents.

Karttunen. Cet algorithme est en fait une généralisation de celui de Kaplan & Kay. L'algorithme est donc le même, si ce n'est qu'au contraire de ses prédécesseurs, Karttunen considère que la règle est par défaut obligatoire, et non optionnelle. De ce fait, Karttunen propose aisément un système tenant compte de règles obligatoires *et* optionnelles, contrairement à Kaplan & Kay : chez Karttunen, le transducteur représentant le remplacement d'une règle optionnelle est simplement l'*union* du remplacement obligatoire et de la cible :

$$(\phi \times \psi) \cup \phi$$

La Figure 5.7 en donne un exemple.

Mohri & Sproat. L'algorithme de Mohri & Sproat est nettement moins coûteux que les précédents, parce qu'il se base sur la définition de trois procédures non coûteuses dédiées aux marqueurs, qui travaillent sur des transducteurs identitaires (cf. Définition 3.2.3) représentant des expressions de la forme $\Sigma^* \alpha$. Notons que les trois procédures veillent à ce que tout état non final de $\Sigma^* \alpha$ devienne final, de sorte que toute string appartenant à Σ^* soit acceptée, pour autant qu'elle respecte certaines conditions :

1. La première procédure, `MARKER1`, ajoute une transition qui *insère* un marqueur *après* α. Le marqueur permet donc de signaler que la string α a été rencontrée. La Figure 5.8 montre le résultat de cette procédure. Dans la Figure, $[\hat{}a]$ signifie « tous les symboles de Σ sauf a].
2. La seconde procédure, `MARKER2`, ajoute une transition qui *supprime* un marqueur *après* α. Le transducteur résultant n'acceptera un chemin contenant un marqueur que si celui-ci se trouve après α, et supprimera ce marqueur. La Figure 5.9 montre le résultat de cette procédure.
3. La troisième procédure, `MARKER3`, ajoute une transition qui *supprime* un marqueur partout *sauf* à la fin de la string α. Le transducteur résultant acceptera toutes les strings suivies d'un marqueur, sauf α, et supprimera ce marqueur. La Figure 5.10 montre le résultat de cette procédure.

Sur la base de ces procédures, l'algorithme de Mohri & Sproat construit les cinq transducteurs suivants :

1. r, construit à partir de `MARKER1`, signale le début du contexte droit ρ à l'aide du marqueur $>$.

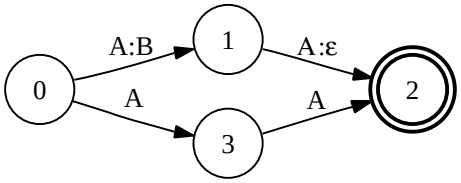


FIG. 5.7: Remplacement optionnel $AA \overset{?}{\rightarrow} B$

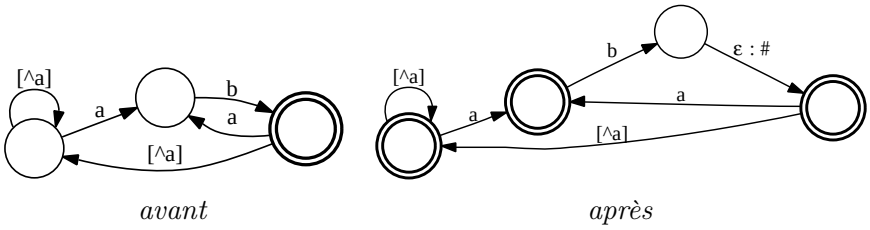


FIG. 5.8: Mohri & Sproat, `MARKER1`, $\alpha = ab$

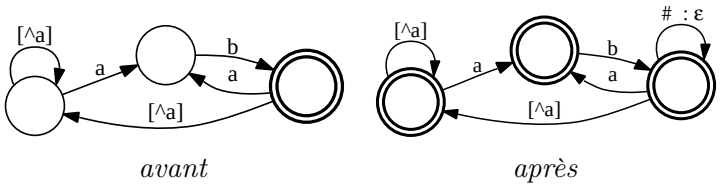


FIG. 5.9: Mohri & Sproat, `MARKER2`, $\alpha = ab$

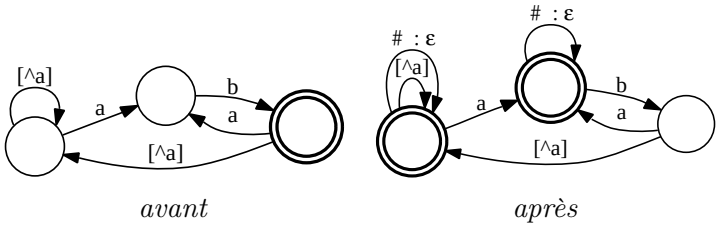


FIG. 5.10: Mohri & Sproat, `MARKER3`, $\alpha = ab$

2. f, construit à partir de `MARKER1`, signale, à l'aide des marqueurs $<_1$ et $<_2$, les occurrences de ϕ qui arrivent juste devant le marqueur $>$, c'est-à-dire devant ρ.
3. *replace* réalise le remplacement $\phi \times \psi$ entre $<_1$ et $>$ et supprime $>$. Sa construction est simple, comme le montre la Figure 5.11, où $\Phi : \Psi$ correspond à $(\phi \times \psi)_{<_1:\epsilon,<_2:\epsilon,>:\epsilon}$ [2].
4. l_1, construit à partir de `MARKER2`, n'accepte $<_1$ que lorsqu'il est précédé de λ, et supprime $<_1$ par la même occasion.
5. l_2, construit à partir de `MARKER3`, n'accepte $<_2$ que lorsqu'il n'est pas précédé de λ, et supprime $<_2$ par la même occasion.

Le transducteur correspondant à la règle est ensuite obtenu par la composition de ces transducteurs dans l'ordre suivant :

$$r \circ f \circ replace \circ l_1 \circ l_2 \tag{5.3.3.2}$$

Mohri & Sproat sont en outre les seuls à proposer une extension de leur algorithme qui permet de définir des règles pondérées. Cette extension consiste à considérer ψ comme une série rationnelle projetant les strings d'un monoïde libre sur un semi-anneau de poids. Le résultat de la compilation d'un règle pondérée donne dès lors un *transducteur pondéré*. Etant donné que l'objectif des règles pondérées est de permettre un *choix* entre plusieurs possibilités, en favorisant la solution *la plus probable* et donc *la moins coûteuse*, les poids pris en compte par Mohri & Sproat sont ceux définis sur le semi-anneau $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$: les poids, qui sont des distances ou des probabilités représentées sous la forme de logarithmes négatifs, sont additionnés le long des chemins du transducteur. Le chemin le moins coûteux sera celui de poids minimal (cf. Section 4.5).

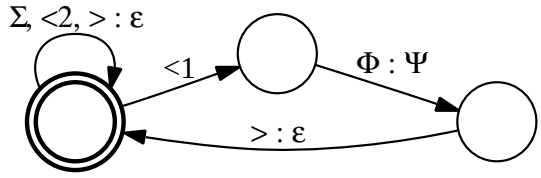


FIG. 5.11: Mohri & Sproat, construction de *replace*

[2] Nous reprenons ici la notation de Mohri, qui identifie par α_x l'automate représentant l'expression régulière α pouvant présenter des occurrences de x à n'importe quelle position.

5.4 Synthèse

A partir d'une expression régulière, description algébrique permettant de représenter de manière concise un langage régulier, il est possible de construire de manière linéaire en terme de temps et d'espace un ϵ-NFA équivalent. Pour ce faire, tout automate intervenant dans la construction du ϵ-NFA doit respecter certaines conventions de représentation, et la mise en œuvre des opérations régulières qui combinent ces automates doit également respecter un certain mécanisme de construction. L'ϵ-NFA obtenu peut ensuite être déterminisé et minimisé.

A partir d'une règle de réécriture, qui décrit une relation entre langages réguliers, un transducteur équivalent peut être construit. Le principe de construction est d'identifier sans ambiguïté les différentes parties de la règle à l'aide de plusieurs transducteurs intermédiaires, qui sont composés ensemble afin de créer le transducteur qui réalise le remplacement prévu par la règle, exclusivement dans le contexte désiré. Les règles de réécriture peuvent être pondérées ; dans ce cas, le résultat de la compilation est un transducteur pondéré.

Ces descriptions syntaxiques complètent donc les machines à états finis pour former un ensemble efficace de modélisation des langages réguliers.

Chapitre 6

Les outils développés

Les applications des machines à états finis en synthèse de la parole, que nous présentons dans la suite de ce document, reposent sur une nouvelle bibliothèque de machines à états finis ainsi que sur un compilateur de règles de réécriture pondérées, de langages réguliers et de dictionnaires.

La Section 6.1 présente les concepts et les principes de notre bibliothèque de machines à états finis, et la Section 6.2 décrit les possibilités offertes par notre compilateur, nommé Ovide.

6.1 La bibliothèque de Machines à Etats Finis

6.1.1 Pourquoi une nouvelle bibliothèque ?

La bibliothèque de machines à états finis nécessaire à nos applications devait respecter trois contraintes :

1. Permettre la définition et la manipulation d'automates et de transducteurs classiques et pondérés.
2. Disposer d'un compilateur autorisant la description des machines désirées sous la forme d'expressions régulières et de règles de réécriture, pondérées ou non.
3. Etre utilisable sur plateforme embarquée, parce que certaines applications, qui peuvent tirer avantage de la puissance des machines à états finis, s'exécutent sur PDA. C'est le cas de la correction orthographique en reconnaissance de caractères (cf. Partie III, Chapitre 16).

De nombreuses bibliothèques de machines à états finis sont téléchargeables sur internet. Nous en dressons un inventaire en Annexe A. Cependant, aucune de ces bibliothèques ne remplit les trois conditions énumérées ci-dessus. Parmi les bibliothèques répertoriées, certaines permettent la définition d'automates et de transducteurs classiques et pondérés.

Quelques unes, parmi celles-ci, sont accompagnées d'un compilateur capable de gérer des expressions et des règles de réécriture pondérées. De ce sous-ensemble de bibliothèques, aucune, cependant, n'est utilisable sur plateforme embarquée, pour l'une des raisons suivantes :

- Soit le code est écrit dans un langage interprété (Prolog, par exemple), ce qui nuit à son efficacité et en limite la portabilité.
- Soit la bibliothèque est uniquement disponible sous la forme d'un ensemble d'exécutables pré-compilés (souvent seulement pour Linux, parfois également pour Windows).
- Soit le code, écrit en C++, recourt massivement aux Templates et au standard STL [1]. Les Templates C++, qui constituent la base du standard STL, définissent des conteneurs génériques pour C++, qui facilitent considérablement le développement, mais ne sont pas bien gérés par les compilateurs pour plateforme embarquée. En outre, le caractère générique des Templates, s'il facilite le développement, nuit à l'efficacité du code. Nous revenons sur la notion de *Template* en Section 6.1.3.1.

Sur la base de ce constat, nous avons pris la décision de redéfinir une bibliothèque complète, mais également un compilateur capable de produire le format de machines accepté par la bibliothèque.

Le choix du langage de la bibliothèque et le mode de représentation des machines ont été déterminés de manière à assurer la portabilité du code sur plateforme embarquée. Le développement de ces outils, en outre, a été l'occasion de proposer certaines *extensions*, d'une part aux représentations classiques des machines à états finis, et d'autre part, aux principes de conception des règles de réécriture. Enfin, afin d'assurer la compatibilité du compilateur et de la bibliothèque, le compilateur lui-même utilise la bibliothèque pour générer les machines correspondant aux expressions et aux règles à compiler.

6.1.2 Principes d'implémentation

Outre la contrainte de portabilité, les deux grands axes qui ont déterminé le développement de notre bibliothèque sont l'efficacité et la conception orientée objet.

6.1.2.1 Efficacité

Notre bibliothèque est un outil : elle doit faciliter la mise en œuvre d'applications rapides et légères. Le langage utilisé pour l'implémentation de la bibliothèque doit donc en assurer l'efficacité.

Dans le cadre de cette thèse par exemple, le domaine d'utilisation est la synthèse de la parole. Or, la synthèse est de plus en plus employée dans le cadre d'applications interactives, comme les systèmes d'information sur bornes, les répondeurs téléphoniques ou les systèmes de lecture pour personnes visuellement handicapées. Le caractère interactif

[1] *Standard Template Library*. Voir www.sgi.com/tech/stl/.

de ces applications nécessite que la synthèse réponde rapidement, voire instantanément à l'utilisateur.

6.1.2.2 Conception orientée objet

Afin de faciliter l'utilisation de notre bibliothèque, nous avons désiré mettre en œuvre les concepts de la programmation orientée objet. La programmation objet (Gunter & Mitchell 1994) consiste à créer une représentation informatique d'un élément en isolant ses données et les fonctions qui les utilisent. L'idée est de créer une structure de données gérée par des méthodes dédiées en respectant certains principes :

L'encapsulation. Les attributs de la structure de données sont cachés et inaccessibles à l'utilisateur de manière directe. L'accès à un attribut (en lecture ou en écriture) n'est possible qu'au travers de méthodes, pour autant qu'elles soient définies et accessibles. Ceci amène à définir la notion d'*interface* : l'interface d'un objet est l'ensemble des méthodes accessibles à l'utilisateur. Si une méthode définie sur l'objet n'est accessible qu'aux autres méthodes de l'objet, elle est dite *privée*.

Le polymorphisme. L'objet peut appartenir à plus d'un type informatique. Par *type*, on entend les types prédéfinis (entier, caractère, booléen, etc.), mais également tout type défini par le programmeur. En proposant d'utiliser un même nom de méthode pour plusieurs types différents, le polymorphisme permet une programmation beaucoup plus générique : l'utilisateur n'a pas à savoir, lorsqu'il utilise une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémente la méthode.

L'héritage. Un objet peut hériter des propriétés d'un autre objet. Ceci évite de redéfinir des méthodes identiques pour des objets similaires. L'héritage implique lui-même la notion de *méthode virtuelle*, qui a un comportement *par défaut*, qui peut être modifié ou complété par l'objet-héritier si nécessaire.

6.1.3 Choix du langage et application des principes d'implémentation

Parmi les langages orientés objet, les plus répandus sont certainement C++ et Java. Ces langages sont cependant inadaptés à l'une ou l'autre des contraintes que nous avons mentionnées. Ce constat nous a poussé à préférer C, langage qui n'est pas orienté objet, mais dans lequel les concepts de l'orienté objet peuvent être mis en œuvre.

6.1.3.1 Inadéquation de C++ et de Java

C++. Développé en 1980 chez AT&T (Stroustrup 2000), C++ a été pensé comme une amélioration du C, auquel il ajoute entre autres :

1. Des opérateurs et des mots-clefs dédiés à la programmation objet.

2. La possibilité de définir des patrons ou *Templates*. Un Template est une définition générique d'une fonction ou d'une classe d'objets qui permet à l'utilisateur d'appliquer un même ensemble d'opérations sur des types informatiques différents. Par exemple, une fonction d'addition

   ```
   T Sum(T v1, T v2)
   ```

   pourrait être définie en Template et accepter en paramètres deux entiers ou deux réels, retournant leur somme dans les deux cas. Les Templates C++ sont au fond une extension de la notion de polymorphisme.

   Les Templates ont permis la construction de la bibliothèque STL, que nous avons mentionnée précédemment et qui propose à l'utilisateur de nombreux conteneurs prédéfinis au format Template : vecteurs, matrices, tables de hashage, arbres, etc. La bibliothèque STL permet à l'utilisateur de tester rapidement *une idée*, sans se soucier de détails d'implémentation, comme le développement de structures de données complexes.
3. Un mécanisme de gestion des erreurs plus performant que les habituels codes d'erreurs généralement utilisés.

Malgré ces avantages, C++ ne pouvait convenir à l'implémentation de notre bibliothèque, et ce pour deux raisons :

1. Comme nous l'avons mentionné, la bibliothèque STL n'est pas acceptée par les compilateurs pour plateforme embarquée. Or, au-delà de l'enrobage orienté objet, le véritable avantage du C++ est la bibliothèque STL.
2. La norme C++ est encore en évolution. D'un compilateur à l'autre, d'une plateforme à l'autre, le même code n'est donc pas assuré de compiler.

Java. Développé chez Sun Microsystems (Naughton 1996), ce langage est apparu en 1995. Son objectif est de pallier les inconvénients du C++ : outre la portabilité difficile de ce langage que nous avons nous-même constatée, les auteurs de Java reprochent au C++ sa gestion manuelle de la mémoire. Il est vrai que C++, à l'instar du C, demande au programmeur de prévoir la désallocation des variables allouées dynamiquement. Java est dès lors un langage :

1. Orienté objet
2. Portable
3. Qui offre un système, le *garbage collector*, chargé de collecter et de désallouer les zones de mémoire allouées dynamiquement.

Java est cependant inadapté à l'impératif d'*efficacité* de notre bibliothèque : ce langage est portable parce qu'il est *semi*-compilé. Le bytecode qui résulte de la compilation est *interprété* par une machine virtuelle en cours d'exécution, ce qui ralentit indéniablement le processus.

6.1.3.2 Recours au C

Nous avons choisi C, bien que ce langage ne soit pas orienté objet, parce qu'il assure nos objectifs de portabilité et d'efficacité. Les concepts de l'orienté objet peuvent en outre y être mis en œuvre, moyennant le respect de quelques conventions simples.

Portabilité. Pour autant que l'on respecte la norme ANSI [2], le langage C est reconnu de manière identique par les compilateurs les plus répandus, tels que *gcc* [3] sous Linux et Windows, et les compilateurs intégrés de *Visual Studio* [4] sous Windows et sous plateforme embarquée. Le même code peut donc être compilé sur l'ensemble des plateformes qui nous intéressent : Windows (2000, XP, Vista), Linux et Pocket Mobile (3.0, 2005).

La portabilité du code sur plateforme embarquée demande en outre que les valeurs numériques soient représentées en point fixe et non en virgule flottante. Comme tous les langages, C autorise la représentation en point fixe. Nous détaillons la méthode utilisée en Section 6.1.4.

Efficacité. Le résultat de la compilation du C est un fichier binaire directement exécuté par la machine, sans interprétation. Ce langage offre en outre les allocations et désallocations manuelles. Contrairement aux auteurs de Java, nous considérons ce type de gestion de la mémoire comme un avantage :

1. *Allocation dynamique.* La gestion manuelle de l'allocation permet d'optimiser le mode d'allocation en fonction du processus. En effet, selon les cas, il est préférable d'allouer la totalité de la mémoire nécessaire en une seule fois ou d'augmenter progressivement la mémoire allouée.
2. *Désallocation dynamique.* La gestion manuelle de la désallocation permet de désallouer au plus tôt toute structure de données qui n'est plus utile au processus. Ceci permet de réduire la mémoire utilisée par le processus et d'éviter le recours à la mémoire virtuelle pour certains processus gourmands. La mémoire virtuelle est une partie du disque dur qui augmente virtuellement l'espace mémoire disponible. Le recours à la mémoire virtuelle ralentit donc le processus, étant donné la lenteur des accès au disque dur.

Bien sûr, une gestion manuelle est risquée, puisqu'il est toujours possible d'utiliser de la mémoire non allouée ou d'oublier de désallouer de la mémoire allouée. Une certaine rigueur de programmation et le recours à des outils de purification du code [5] permettent cependant de dépasser ces limites fort aisément.

[2] Voir www.ansi.org/.
[3] Voir gcc.gnu.org/.
[4] Voir msdn2.microsoft.com/en-us/vstudio/default.aspx.
[5] Par exemple, *Rational Purify* : www-306.ibm.com/software/awdtools/purify/.

Conception orientée objet. En C, nous proposons de gérer les principes de l'orienté objet comme suit (cf. Tables 6.1 et 6.2 pour un exemple) :

• *Méthode d'encapsulation.* En C, l'encapsulation peut être réalisée en séparant les déclarations entre le fichier entête (`.h`) et le fichier source (`.c`). Le fichier entête, qui est le fichier accessible à l'utilisateur, ne contient que les informations qui peuvent être connues de l'utilisateur, c'est-à-dire l'interface de l'objet. Le fichier source contient quant à lui toutes les informations privées de l'objet, telles que ses membres et ses méthodes privées.

Dans le fichier-entête (Table 6.1), la structure `_Object` est simplement déclarée. Dans le fichier-source (Table 6.2), les membres de la structure sont définis, et la méthode privée `Object_Private` est déclarée et définie.

• *Polymorphisme.* Notre bibliothèque autorise deux types de machines, les machines dynamiques et les machines binaires. Les machines dynamiques sont créées transition après transition par l'utilisateur ou par l'application d'une opération (par exemple, la composition) sur une ou plusieurs machines. Les machines binaires sont des machines *précalculées* auxquelles on n'ajoutera plus de transitions, et sont dès lors figées dans un format compact. Le polymorphisme permet à l'utilisateur d'employer les mêmes opérations sur les deux types de machines sans avoir besoin de connaître le type des machines manipulées. Dans notre implémentation, le type de l'objet est déterminé *à l'exécution*, lors de sa création. On parle de *typage dynamique*. Notre typage dynamique consiste à instancier dynamiquement les méthodes de l'objet : la structure de données représentant l'objet contient des pointeurs sur méthode, que l'on fait pointer vers les méthodes désirées une fois le type connu.

Dans le fichier-source (Table 6.2), on commence par déclarer des interfaces de méthode (`METH1` et `METH2`). Ces interfaces permettent de déclarer des pointeurs sur méthode dans la structure `_Object`. La déclaration d'une interface de méthode permet de contraindre un pointeur à n'accepter que les méthodes respectant cette interface. Ces pointeurs sont initialisés par la méthode `Object_InitMethod` à la création de l'objet, après avoir identifié le type de l'objet créé (ici, créé à partir d'un fichier).

• *Héritage.* Dans notre bibliothèque, l'héritage est implicite. En effet, certaines méthodes, qui ne dépendent pas du type de la machine chargée, sont implémentées une seule fois et ne nécessitent pas de pointeurs sur méthode. C'est le cas des méthodes `Object_Create` et `Object_InitMethod` (Table 6.2).

• *Structure de données.* Dans un langage orienté objet, les méthodes définies sur l'objet ont un argument implicite qui est la structure de données elle-même. En C++ par exemple, la structure est d'ailleurs accessible dans la méthode *via* le pointeur `this`. Cela n'existe pas en C. Pour pallier ce manque, nos méthodes prennent comme premier argument un pointeur sur la structure de données. En interne, ce pointeur est toujours appelé `pThis`, de manière à faire référence à la notion C++. Notons que les méthodes qui créent l'objet ne prennent pas ce pointeur en argument, mais le retournent après l'avoir alloué.

```
typedef struct _Object Object;
Object* Object_Create(const char* file);
void    Object_Destroy(Object* pThis);
int     Object_Process(Object* pThis, char var1);
```

Fichier « objectdyn.h »

```
typedef struct _ObjectDyn ObjectDyn;
int     ObjectDyn_Process(void* pThis, char var1);
void    ObjectDyn_Private(void* pThis, char var1, int var2);
```

Fichier « objectbin.h »

```
typedef struct _ObjectBin ObjectBin;
int     ObjectBin_Process(void* pThis, char var1);
void    ObjectBin_Private(void* pThis, char var1, int var2);
```

TAB. 6.1: Programmation orientée objet en C – entêtes ("object.h")

Quelques données. Développée selon la norme ANSI en respectant le modèle orienté objet présenté, la bibliothèque compte environ 45 000 lignes de code. Elle est actuellement disponible sous Windows (2000 et XP), Linux et Pocket Mobile (3.0 et 2005).

6.1.4 Représentation générale des machines

Dans notre bibliothèque, une machine à états finis est une structure :

`FSM = (alpha1, alpha2, pFSM)`

Elle possède donc deux alphabets (`alpha1`, `alpha2`), et un pointeur sur la véritable machine (`pFSM`), dynamique ou binaire selon le cas. Que la machine soit dynamique ou binaire, elle possède en substance un vecteur d'états, dont un seul est initial. Chaque état connaît les transitions qui le quittent, mais pas celles qui l'atteignent.

Pour un programme appelant, une transition a toujours la structure suivante :

`Trans = (symbol1, symbol2, weight, stateN)`

où `symbol1` est le symbole d'entrée, `symbol2` est le symbole de sortie, `weight` est le poids et `stateN`, l'état suivant. Notons que les quatre éléments de la transition sont des entiers non signés.

Cette représentation nécessite quelques commentaires.

6.1.4.1 Transducteurs pondérés

Les machines ont toujours la structure de *transducteurs pondérés* (cf. Section 4.3). Or, la bibliothèque permet de représenter des automates et des transducteurs, pondérés et non pondérés. Ceci est dû au fait que :

- Le semi-anneau (cf. Section 4.2.1) sur lequel est définie la bibliothèque est le semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$: les poids sont toujours positifs, et l'élément neutre est 0. Les machines non pondérées sont donc simplement des machines dont les poids des transitions sont à 0.
- Un transducteur peut toujours être considéré comme un automate via son automate sous-jacent (cf. Définition 3.2.2), ce qui autorise l'applications aux transducteurs des opérations définies sur les automates.
- Un transducteur dont les symboles d'entrée sont équivalents aux symboles de sortie n'est jamais qu'un automate, mais que l'on qualifie de *transducteur identitaire* (cf. Définition 3.2.3). Nos automates sont représentés sous la forme de transducteurs identitaires.

Quel est l'intérêt de cette représentation constante ? Elle assure une *compatibilité* des machines, et évite dès lors de nombreuses conversions

$$\begin{array}{c} \textit{automate} \leftrightarrow \textit{transducteur} \\ \textit{pondéré} \leftrightarrow \textit{non pondéré} \end{array}$$

coûteuses en terme de réallocations dynamiques. Par exemple,

- La *composition* ne doit jamais convertir un automate en son transducteur identitaire, puisque le transducteur identitaire *est* la représentation de l'automate. L'algorithme adopte donc toujours le même comportement : étant donné deux FSMs F_1 et F_2, la composition $F_1 \circ F_2$ consiste à comparer, pour toute transition, la sortie de F_1 à l'entrée de F_2, et, en cas d'intersection, à créer une transition avec l'entrée de F_1 et la sortie de F_2. Si les deux FSMs sont des automates, l'opération revient à une simple intersection, ce qui correspond bien au principe de la composition, généralisation de l'intersection (cf. Section 3.3.3).
- Lors d'une *première projection* (cf. Définition 3.2.6), on recopie simplement, pour toute transition, le symbole d'entrée sur le symbole de sortie. La seconde projection est le procédé inverse.

Il est cependant utile de savoir si la machine est un automate ou un transducteur. Par exemple, la minimisation d'un transducteur devra présenter une étape de préfixation (cf. Section 3.4.3) de sa seconde projection, alors que cette étape pourra être épargnée à l'automate. Un attribut de la machine, `FSMType`, permet donc de savoir si la machine est un automate ou un transducteur, et est mis à jour si nécessaire par les différentes opérations. Par exemple, une projection changera la valeur de `FSMType`, de *transducteur* à *automate*.

```
#include "object.h"
#include "objectdyn.h"
#include "objectbin.h"
#define DYN 1
#define BIN 2
typedef int  (*METH1)(void*, char);
typedef void (*METH2)(void*, char, int);
/* declaration */
struct _Object {
    void* pObject;
    char  cMember1;
    int   iMember2;
    METH1 pMethProcess;
    METH2 pMethPrivate;
};
void Object_Private(Object* pThis, char var1, int var2);
void Object_InitMethod(Object* pThis, int iMode);
/* definition */
Object* Object_Create(const char* file) {
    Object* pThis = /* allocation */;
    int iMode = /* read file, create pObject, return DYN|BIN */;
    Object_InitMethod(pThis, iMode);
    return pThis;
}
int Object_Process(Object* pThis, char var1) {
    return pThis->pMethProcess(pThis->pObject, var1);
}
void Object_Private(Object* pThis, char var1, int var2) {
    pThis->pMethPrivate(pThis->pObject, var1, var2);
}
void Object_InitMethod(Object* pThis, int iMode) {
    switch(iMode) {
    case DYN :
        pThis->pMethProcess = ObjectDyn_Process;
        pThis->pMethPrivate = ObjectDyn_Private;
        break;
    case BIN :
        pThis->pMethPrivate1 = ObjectBin_Process;
        pThis->pMethPrivate2 = ObjectBin_Private;
        break;
    default :
        /* error */
    }
}
```

TAB. 6.2: Programmation orientée objet en C – source ("object.c")

6.1.4.2 Symboles numériques

Les symboles sont numériques dans les machines. La bibliothèque a donc besoin d'un système de conversion permettant de passer des alphabets aux valeurs numériques, et inversement. Les objets `alpha1` et `alpha2` réalisent ces conversions.

6.1.4.3 Poids entiers

Les poids sont des entiers, nous n'utilisons pas de réels. Ceci signifie qu'*a priori*, nous perdons la partie décimale du poids. Cependant, dans notre bibliothèque, un poids est multiplié par 10 000, avant d'être tronqué à sa partie entière. Ceci permet de conserver les quatre premières décimales du poids, et donc de garder une certaine précision. Nous employons donc une représentation « point fixe » très simple. Cependant, dans le cas des valeurs manipulées par la bibliothèque, cette représentation est suffisante, puisque les poids manipulés sont des logarithmes négatifs de probabilités ou des distances entières.

6.1.4.4 Etats finaux

On constate que la machine ne possède pas d'ensemble d'états finaux. Cependant, chaque état final possède une transition supplémentaire, dont l'élément `stateN` vaut la plus grande valeur en entier non signé (`0xFFFFFFFF`), réservée pour la constante `FINAL`. Si l'état final a un poids, le poids de cette transition sera différent de 0.

6.1.5 Format binaire

Nous avons mentionné que les FSMs et les alphabets peuvent être représentés et sauvegardés au format binaire. Le but de cette représentation est double :

1. Limiter la place mémoire requise (et non la place disque, qui importe moins).
2. Limiter le temps de chargement des machines, qui peuvent parfois contenir plusieurs millions d'états.

Un *dump* de l'objet à sauver permet de respecter ces contraintes. Le dump – le *cliché-mémoire* en français – est dans sa définition originale [6] l'opération qui consiste à sauver sur le disque dur une impression de l'état de la mémoire à un moment donné, le plus souvent à des fins de débogage. Sur les systèmes UNIX, le dump a longtemps permis d'examiner *l'état* de la mémoire.

La définition que nous donnons du dump est fort similaire : le dump d'un objet est l'impression sur le disque dur d'une représentation *optimale* de cet objet en mémoire. Dans cette définition, le dump ne sauve donc pas l'état *réel* de la mémoire, mais calcule la place minimale requise pour représenter l'objet *en l'état*. Le dump implique donc que l'objet

[6] On parle en fait de *Core Dump*.
Voir www.scit.wlv.ac.uk/cgi-bin/mansec?4+core

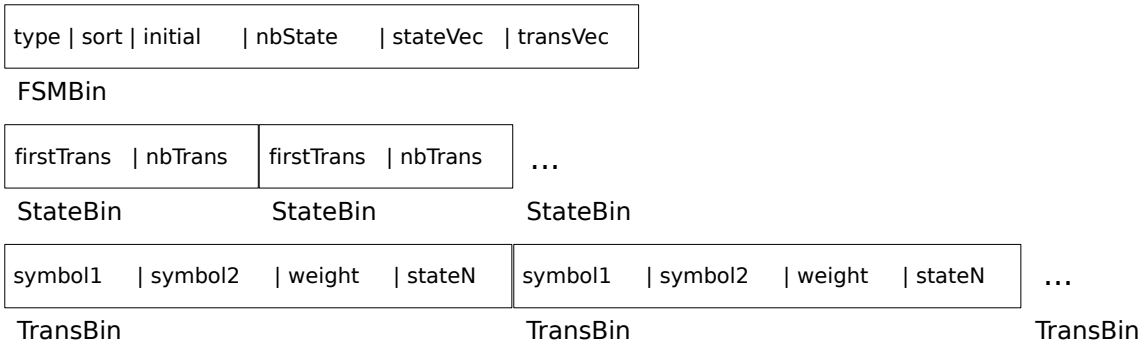


FIG. 6.1: Représentation binaire d'un FSM

est considéré comme *figé* et ne sera plus modifié par la suite, parce que la place minimale requise pour le représenter n'autorisera plus de modifications.

Le dump doit dès lors (1) sauver les données sans discontinuité et (2) intégrer dans la sauvegarde les pointeurs de l'objet et les valeurs de ces pointeurs. La valeur sauvée d'un pointeur ne peut cependant être directement l'adresse d'une case mémoire, parce que le lieu de la mémoire où est chargé un objet varie inéluctablement d'une instanciation du programme à l'autre. Par contre, cette valeur peut être *relative* à un point donné de la mémoire : par exemple, l'intervalle séparant l'endroit où l'objet commence en mémoire de l'endroit où se trouve la donnée indiquée par le pointeur.

La Figure 6.1 donne un exemple d'architecture binaire pour une machine à états finis. Pour des raisons de lisibilité, le vecteur est présenté sur plusieurs lignes, mais il faut l'imaginer ininterrompu. On constate que les données du FSM incluent les deux pointeurs `stateVec` et `transVec`. La valeur du pointeur `stateVec` est l'intervalle, en nombre d'octets, qui sépare le début du FSM du premier état. En l'occurrence,

$$\texttt{stateVec = sizeof(FSMBin) ;}$$

où `sizeof` est une fonction C qui retourne la taille d'un type en octet. De même, la valeur du pointeur `transVec` est l'intervalle, en nombre d'octets, qui sépare le début du FSM de la première transition. Cette valeur inclut donc la structure FSM et les états :

$$\texttt{transVec = sizeof(FSMBin) + nbState} \times \texttt{sizeof(StateBin) ;}$$

Comme le montre la Figure 6.1, un état possède une valeur `firstTrans`. Il s'agit de l'*index* de la première transition de l'état dans le vecteur de transitions.

Le chargement de l'objet dumpé devient dès lors très simple et très rapide. Il contient deux étapes :

1. Un vecteur d'octets de la taille du fichier binaire est alloué en un seul bloc, et le contenu du fichier y est copié.

2. Chaque pointeur de l'objet est mis à jour : à sa valeur est additionnée l'adresse du début du vecteur. Concernant l'objet FSM, et considérant que l'objet a été chargé dans le vecteur d'octets `fileVec`, cette mise à jour vaut :

```
stateVec = (StateBin*) (fileVec + (unsigned long) stateVec);
transVec = (TransBin*) (fileVec + (unsigned long) transVec);
```

   La valeur du pointeur (resp. `stateVec` ou `transVec`) est castée en entier long non signé afin de récupérer (en nombre d'octets) l'intervalle sauvé. Cet intervalle est additionné au pointeur `fileVec`, de manière à pointer le début de la structure désirée. Cependant, le résultat pointe de l'octet ; il est de ce fait lui-même casté dans le type du pointeur (resp. `StateBin` ou `TransBin`).

Le mode de chargement assure que la taille en mémoire de l'objet dumpé est identique à la taille du fichier binaire sauvé sur le disque dur. Dans les chapitres suivants, nous donnons des exemples de machines binaires et nous comparons les deux modes de représentation au niveau de la place mémoire occupée et du temps nécessaire au chargement.

6.1.6 Principales méthodes disponibles sur les FSMs

La liste ci-dessous ne présente que les méthodes les plus significatives.

Création. Il est possible de créer des machines dynamiques, binaires ou retardées.

- `FSM_Create` : les machines dynamiques et binaires peuvent être créées à partir d'un fichier : fichier texte dans le cas d'une machine dynamique, fichier binaire dans le cas d'une machine binaire.
- `FSM_CreateDyn` : les machines dynamiques peuvent également être créées transition après transition, par l'utilisateur.
- `FSM_CreateLazy` : une machine retardée met en œuvre le principe de l'*évaluation retardée* (ou *paresseuse*, traduction littérale de *lazy algorithm* en anglais). Il s'agit d'une technique de programmation où une opération n'est réalisée que lorsque les résultats de cette opération sont réellement *nécessaires*. Cette technique peut être utilisée à des fins d'optimisation : elle permet d'éviter de calculer un résultat qui pourrait ne pas être utilisé. L'évaluation retardée est réalisable sur une machine à états finis à partir du moment où l'algorithme n'a pas besoin de parcourir toute la machine, mais peut prendre une décision locale (Mohri *et al.* 1996, 2000, 2002, Mohri & Riley 2002). C'est le cas de :
  - la suppression-ϵ,
  - la déterminisation,
  - la projection,
  - la composition

dont les algorithmes ne traitent qu'un état à la fois. Par contre, une opération comme la minimisation, qui compare les états afin de prendre une décision quant à leur équivalence, ne peut proposer une évaluation retardée.

Sauvegarde. Une machine peut être sauvée au format texte (`FSM_Print`) ou au format binaire (`FSM_Dump`).

Il est également possible de dessiner une machine (`FSM_Draw`) au format du logiciel *Graphviz*, développé par AT&T [7]. Les machines qui illustrent ce document ont été réalisées avec cette méthode.

Informations. D'un FSM, il est possible de connaître :

- `FSM_GetMode` : le mode de création (`DYN/BIN`),
- `FSM_GetType` : le type de machine (`FSA/FST`),
- `FSM_GetSort` : le mode de classement des transitions. Les transitions sont indexées selon un ou plusieurs critères parmi le symbole d'entrée, le symbole de sortie, le poids et l'état suivant. Le classement est important parce qu'il détermine le temps d'accès aux informations nécessaires aux algorithmes. La composition, par exemple, a besoin que la machine d'entrée soit classée sur le couple (*sortie, poids*), et que la machine de sortie soit classée sur le couple (*entrée, poids*). Les algorithmes modifient le mode de classement au besoin.
- `FSM_GetLevel` : le niveau de représentation (du ϵ-NFA à la machine minimale). Ce niveau de représentation est vérifié lors de tout ajout d'une nouvelle transition. Par exemple, si une transition (ϵ,ϵ) est ajoutée, le FSM passe automatiquement au niveau ϵ-NFA,
- `FSM_GetInitial` : l'état initial,
- `FSM_GetNbState` : le nombre d'états,
- `FSM_GetNbTrans` : le nombre de transitions d'un état,
- `FSM_GetStateWeight` : le poids d'un état.

Modifications. Sur les machines dynamiques, il est possible d'utiliser :

- `FSM_AddTrans` : ajout d'une transition à un état,
- `FSM_SetInitial` : définition de l'état initial,
- `FSM_SetFinal` : ajout d'une transition finale pondérée à un état,
- `FSM_SetNoFinal` : suppression de la transition finale d'un état.

[7] Voir www.graphviz.org/.

Accès aux transitions. Il est possible d'accéder à une transition ou à l'ensemble des transitions d'un état :

- `FSM_GetTrans` : pour un état donné, obtention d'un pointeur sur une transition dont on spécifie l'indice,
- `FSM_GetAllTrans` : pour un état donné, obtention d'un pointeur sur l'ensemble de ses transitions.

Opérations. La bibliothèque ne contient que les versions pondérées des algorithmes définis sur les machines à états finis, puisque toutes nos machines sont au moins pondérées par l'élément neutre du semi-anneau tropical (0) :

- `FSM_EpsilonFree` : la suppression-ϵ,
- `FSM_Determinize` : pour un transducteur, il s'agit d'une séquentialisation.
- `FSM_Minimize` : pour un transducteur, la préfixation de la seconde projection est réalisée.
- `FSM_Concat`.
- `FSM_Union`.
- `FSM_Closure` : étoile de Kleene.
- `FSM_Complement`.
- `FSM_Inverse`.
- `FSM_Reverse`.
- `FSM_Project` : première ou seconde projection, selon l'argument.
- `FSM_Compose`.
- `FSM_GetBestPath` : recherche des n meilleurs chemins, n étant passé en argument.

6.1.7 Principales méthodes disponibles sur les alphabets

La liste ci-dessous ne présente que les méthodes les plus significatives.

Création. Un alphabet peut être créé de manière dynamique ou binaire.

- `Alpha_CreateEx` : un alphabet de type prédéfini peut être déclaré sans recours à un fichier. Les types prédéfinis sont `ASCII` (256 symboles), `ALPHA` (les lettres de l'alphabet latin en minuscule et en majuscule), `NUM` (de 0 à 9) et `ALPHANUM` (combinaison des deux précédents).
- `Alpha_Create` : les alphabets dynamiques et binaires peuvent être créés à partir d'un fichier : fichier texte dans le cas d'un alphabet dynamique, fichier binaire dans le cas d'un alphabet binaire. Le fichier peut contenir un alphabet de type prédéfini, mais également un alphabet défini par l'utilisateur, avec un symbole par ligne.

Sauvegarde. Un alphabet peut être sauvé au format texte (`Alpha_Print`) ou au format binaire (`Alpha_Dump`).

Informations. D'un alphabet, il est possible de connaître :

- `Alpha_GetMode` : le mode de création (`DYN/BIN`),
- `Alpha_GetType` : le type d'alphabet (`ALPHA/NUM/...`),
- `Alpha_GetNbSymbol` : le nombre de symboles.

Modifications. Il est possible d'insérer un nouveau symbole dans un alphabet dynamique (`Alpha_Insert`).

Opérations. Concernant les alphabets, les opérations principales disponibles sont :

- `Alpha_Str2Val` : conversion d'un symbole alphabétique en une valeur numérique,
- `Alpha_Val2Str` : conversion d'une valeur numérique en un symbole alphabétique.

6.1.8 Extensions

Au cours du développement des applications que nous présentons dans les parties suivantes de ce document, nous avons constaté que certaines caractéristiques des langages définis pouvaient être exploitées, afin de réduire de manière considérable la place mémoire et la place disque nécessaires à leur représentation.

L'exploitation de ces caractéristiques a donné lieu à la définition des *classes de symboles* et des *graphes orientés pondérés*.

6.1.8.1 Les classes de symboles

Positionnement du problème. Comme le constatent Mohri & Sproat (1996), un langage régulier, qui est le résultat de la compilation de règles de réécriture, est souvent de la forme

$$\Sigma^* \, \alpha$$

où Σ est l'alphabet sur lequel est défini le langage, et α est une expression régulière définie sur Σ.

Pour un langage de ce type, la machine à états finis équivalente, une fois déterminisée et minimisée, est par nature *complète* : chaque état présente une et une seule transition pour chaque symbole de l'alphabet.

De ce fait, la taille d'une machine complète dépend de deux facteurs : le nombre d'états qu'elle contient et, surtout, la taille de l'alphabet sur lequel elle est définie. Une machine complète peut donc rapidement nécessiter une place de représentation considérable, si elle présente de nombreux états et travaille sur un alphabet conséquent.

Que faire dans ce cas ? Il est évidemment possible d'éviter de déterminiser et de minimiser la machine. La machine non déterministe ne sera pas complète, ce qui limitera la place nécessaire à sa représentation. Par contre, le temps de parcours d'une string ne sera plus linéairement proportionnel à la taille de la string, ce qui n'est pas envisageable dans le cadre d'applications qui doivent répondre en temps réel.

Le nombre d'états ne pouvant être réduit, nous proposons un nouveau mode de représentation des transitions : dans nos machines à états finis, une transition peut être étiquetée par une *classe* de symboles. A l'aide des classes de symboles, la place requise pour représenter une machine complète peut être fortement réduite.

Définition 6.1.1 (Classe de symboles)**.** *Dans notre bibliothèque de machines à états finis, une* classe de symboles *regroupe les symboles de toutes les transitions d'un état p qui atteignent un même état q et présentent la même pondération. Une classe de symboles permet de réduire n transitions à une seule transition, étiquetée par un symbole spécial réservé à la classe.*

Identification d'une classe. Dans un FSM, une transition étiquetée par une classe de symboles est identifiable, parce que le symbole numérique qui étiquette la transition est supérieur à la somme des symboles des alphabets du FSM, `alpha1` et `alpha2` [8].

Construction des classes. Les classes d'un FSM sont gérées et créées par un objet spécial, `Class`, construit autour d'une table de hashage. L'algorithme est très simple. Tous les états du FSM sont traités successivement, dans une boucle. Pour un état donné, le principe est de recenser l'ensemble des transitions atteignant un même état q et présentant un même poids. Dans ce cas, une nouvelle classe est construite si la table de hashage ne la contient pas encore, et les transitions concernées sont remplacées par une nouvelle transition de même poids, étiquetée par le symbole de la classe.

Les types de classes. Etant donné que notre bibliothèque permet de manipuler des automates et des transducteurs, différents types de classes peuvent être définis. Le type de la classe dépend toujours des étiquettes des transitions à partir desquelles on construit cette classe :

1. Toutes les étiquettes sont du type (a:a) : le symbole d'entrée est identique au symbole de sortie. La classe correspond dès lors à un simple ensemble. Par exemple :

   $$\{a, b, c, d, e\}$$

[8] Si `alpha1` et `alpha2` sont identiques, le symbole d'une classe est simplement supérieur au nombre de symboles d'`alpha1`.

2. Les étiquettes sont du type (a:b), mais on constate que les symboles d'entrée sont tous projetés sur les mêmes symboles de sortie. Par exemple, a et b en entrée sont projetés sur c, d et e en sortie. Dans ce cas, la classe est une transduction entre deux ensembles :

   $$\{a, b\} : \{c, d, e\}$$

3. Aucune similitude ne peut être dégagée de l'analyse des transitions. Dans ce cas, la classe est un ensemble de paires de symboles. Par exemple :

   $$\{(a{:}a), (b{:}c), (d{:}a), (e{:}b)\}$$

Les deux premiers types de classes utilisent des ensembles de symboles. Tels que nous les avons présentés, ces ensembles sont décrits en *extension* : nous y mentionnons la totalité des symboles qui appartiennent à l'ensemble. Or, dans certains cas, l'ensemble regroupe plus de la moitié des symboles de l'alphabet. Pour plus de compacité, ces ensembles sont décrits en *compréhension*. Par exemple, un ensemble contenant les symboles $\{a, b, c, d\}$ pour un alphabet $\Sigma = \{a, b, c, d, e, f\}$ peut être défini en compréhension, comme étant l'ensemble de tous les symboles qui ne sont ni e, ni f :

$$!\{e, f\}$$

La définition en compréhension est bien sûr fortement dépendante de l'alphabet, sur lequel l'objet `Class` possède un pointeur.

Complexité. L'appartenance d'un élément à un ensemble est déterminée par recherche dichotomique. La complexité du calcul de l'intersection de deux classes C_1, C_2 est donc en $O(|C_1| \log_2(|C_2|))$, où $|C_i|$ représente le nombre d'éléments de la classe i, avec $1 \leq i \leq 2$.

Notons que le temps imparti au calcul de l'intersection est compensé par le fait qu'un état complet, contenant n transitions, est compacté grâce aux classes en un état complet, contenant m transitions, avec $m \ll n$. La composition reste donc efficace, tout en étant calculée sur des machines nettement moins volumineuses.

La Figure 6.2 illustre l'intérêt des classes de symboles. La règle compilée, définie sur l'ensemble des chiffres de 0 à 9, est la suivante :

$$(1|2) \rightarrow (3|4|5) :: 3 \_ 4 \,/\, 1$$

Le graphe de gauche est non compacté, contrairement au graphe de droite, qui présente des classes de symboles.

Utilisation des classes. Seules la composition et les méthodes de sauvegarde des machines (cf. Section 6.1.6) ont été prévues pour gérer les classes de symboles. Pour les autres méthodes (déterminisation, minimisation, complément, etc.), il est nécessaire de préalablement décompacter les machines avant de les traiter.

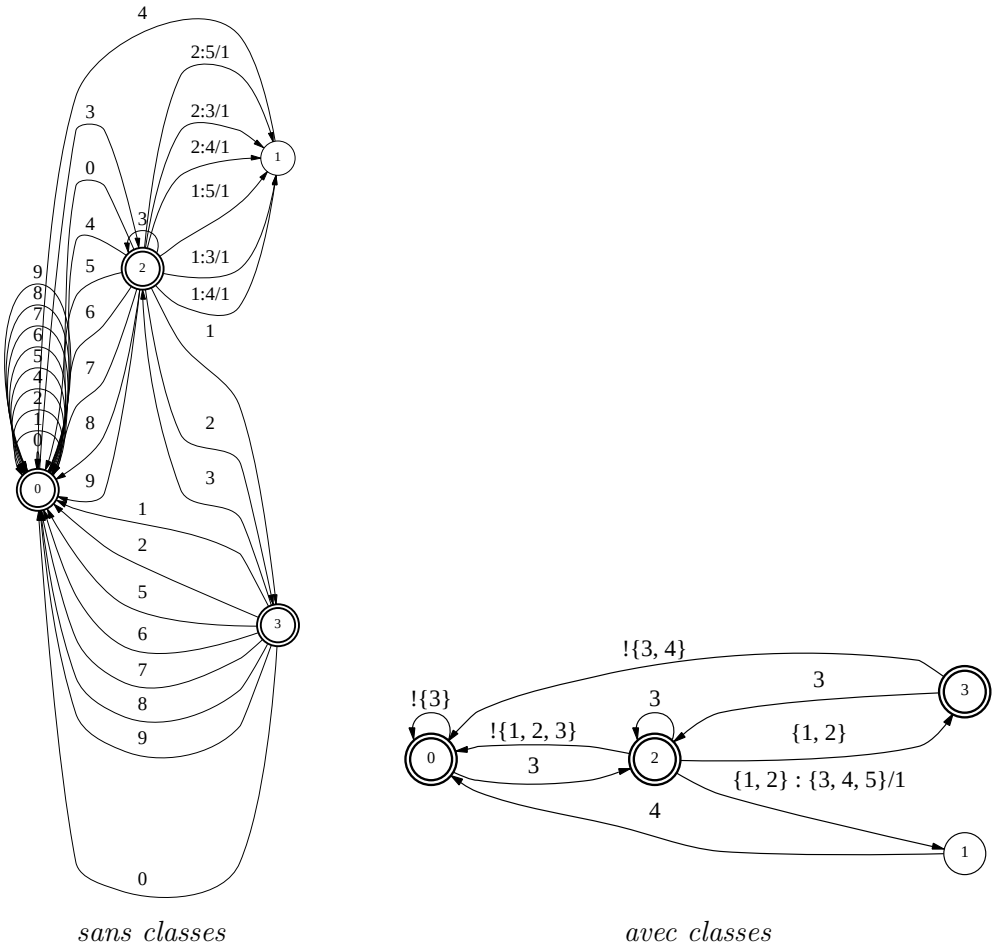


FIG. 6.2: Compaction à l'aide de classes de symboles

De l'ensemble des opérations régulières définies sur les machines à états finis, seule la composition a donc dû être adaptée. Comme nous l'avons détaillé en Section 3.3.3, la composition est une généralisation de l'intersection des automates. Au niveau des transitions, le calcul de l'intersection consiste initialement à déterminer les symboles identiques qui étiquettent les transitions de deux états en cours d'examen. Or, cette simple comparaison n'est plus possible : lorsqu'au moins une des deux transitions comparées est étiquetée par une classe, l'object `Class` doit déterminer s'il y a intersection. Le calcul de cette intersection varie évidemment selon que les deux symboles à comparer sont des classes, ou que seul l'un des symboles est une classe. Dans les deux cas, cependant, le calcul retourne le résultat de l'intersection sous la forme d'une nouvelle classe, prête à être intégrée dans la machine correspondant au résultat de la composition.

Exemple d'utilisation. Dans le cadre de l'analyse morphologique (cf. Section 15.4.2.5), l'un des langages modélisés travaille sur 560 symboles et prend 98 Mo. Compactée à l'aide de classes de symboles, cette machine ne prend plus que 5,5 Mo...

6.1.8.2 Les graphes orientés pondérés

Positionnement du problème. Dans certains cas particuliers, il peut être intéressant de penser un langage sous la forme d'un graphe orienté pondéré. La Figure 6.3 illustre le principe. Dans ce type particulier de langage, un état est attribué à chaque symbole de l'alphabet sur lequel est défini la machine. De ce fait, une transition entre deux états modélise toujours le coût de transition entre les deux symboles qui leur sont attribués. Dans cette modélisation particulière, un état final indique que le symbole qui lui est attribué peut terminer un mot du langage. L'état initial, quant à lui, a ceci de particulier qu'il ne correspond à aucun symbole, mais établit le coût de commencer un mot du langage par un symbole particulier.

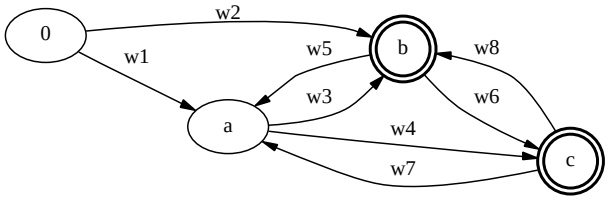


FIG. 6.3: Langage pensé comme un graphe orienté pondéré. L'alphabet est $\{a, b, c\}$. Seuls a et b peuvent être initiaux. Seuls b et c peuvent être finaux. Les transitions sont toutes étiquetées par des poids différents

Sur ce type de machine, la déterminisation et la minimisation sont inefficaces. La déterminisation est inutile, puisque chaque état présente, par construction, au maximum une seule transition pour un symbole donné. La minimisation a par ailleurs un intérêt réduit, parce que rares sont les symboles – et donc les états – qui acceptent le même langage pondéré.

Il ressort de cette analyse que la structure des transitions de notre bibliothèque est, dans le cas d'un graphe, inutilement redondante, puisqu'elle présente systématiquement la même valeur pour le symbole d'entrée, le symbole de sortie et l'état suivant. Ceci est illustré par la Figure 6.4.

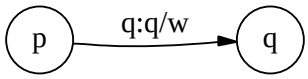


FIG. 6.4: Redondance des transitions d'un graphe : les deux symboles et l'état atteint sont identiques

Nous avons de ce fait décidé d'augmenter le polymorphisme de notre bibliothèque de machines à états finis, de manière à autoriser la représentation de *graphes orientés pondérés*. De manière formelle,

Proposition 6.1.1. *Un automate pondéré $A = (\Sigma, Q, i, F, \delta, \rho)$, peut être représenté par un graphe orienté pondéré $G = (Q, i, F, \delta, \rho)$, équivalent à A, pour autant que :*

(a) $\Sigma = Q$
(b) $\forall p \in Q, \forall q \in \Sigma,$ *si* $\delta(p, q)$ *existe, alors* $\delta(p, q) = q$

Définition 6.1.2 (Propriété de graphe)**.** *Dans notre bibliothèque, une transition* `(symbol1, symbol2, weight, stateN)` *possède la propriété de graphe lorsque* `symbol1` = `symbol2` = `stateN`.

Définition 6.1.3 (Graphe orienté pondéré)**.** *Dans notre bibliothèque, une machine à états finis est un graphe orienté pondéré si et seulement si toutes ses transitions possèdent la propriété de graphe.*

Détection d'un graphe. Le Pseudocode 24 présente l'algorithme de détection d'un graphe. Spécialement défini pour les machines de notre bibliothèque, cet algorithme se contente de vérifier que chaque transition de la machine concernée possède la propriété de graphe. Il ne teste pas l'équivalence des ensembles d'états et de symboles, parce que les états et les symboles des transitions de nos machines sont numériques. En outre, il manipule des transducteurs, puisque les automates, dans notre bibliothèque, sont représentés sous la forme de transducteurs identitaires.

Construction d'un graphe. Dans notre bibliothèque, le graphe n'existe qu'au format binaire. Il serait en effet inutile de considérer comme un graphe une machine dynamique, susceptible d'être complétée à tout moment par de nouvelles transitions.

La construction d'un graphe se fait donc au moment de la sauvegarde binaire, dont la première étape devient l'algorithme de détection d'un graphe. Si la machine est un graphe, le format de sauvegarde est adapté. Deux différences distinguent le format « graphe binaire » du format binaire standard :

1. Les transitions du vecteur de transitions sont maintenant de la forme :

   ```
   GraphTrans = (weight, stateN)
   ```

   Ceci permet l'économie de 2x4 octets par transition.

2. Le type n'est plus `FSA` ou `FST`, mais `GRAPH`.

Chargement d'un graphe. Nous avons étendu le polymorphisme de la bibliothèque : en plus de la structure dynamique et de la structure binaire standard, la bibliothèque possède une structure binaire dédiée au graphe, appelée `FSMGraph`, qui se distingue de la structure

Require: $T = (\Sigma_1, \Sigma_2, Q, i, F, E, \rho)$
Ensure: $IsGraph$ vaut 1 si T est un graphe, 0 sinon
1: $IsGraph \leftarrow 1$
2: $p \leftarrow 0$
3: **while** $p < |Q|$ **and** $IsGraph = 1$ **do**
4: $t \leftarrow 0$
5: **while** $t < |E[p]|$ **and** $IsGraph = 1$ **do**
6: $e \leftarrow E[p][t]$
7: **if** $e.a_1 \neq e.a_2$ **or** $e.a_1 \neq e.q$ **then**
8: $IsGraph \leftarrow 0$
9: **end if**
10: $t \leftarrow t + 1$
11: **end while**
12: $p \leftarrow p + 1$
13: **end while**
14: **return** $IsGraph$

Pseudocode 24: Test de détection d'un graphe

binaire classique par ses transitions, de type `GraphTrans`, et par ses méthodes de gestion des transitions.

Au chargement d'une machine, la méthode `FSM_Create` vérifie le type de la machine à charger. Si la valeur est `GRAPH`, la structure binaire initialisée est `FSMGraph`.

Encapsulation des transitions. Avant la définition du graphe, notre bibliothèque possédait un seul format de transition. Il ne nous avait dès lors pas paru utile d'encapsuler la structure correspondante (`Trans`), qui faisait partie de l'interface de la bibliothèque. Toute fonction, interne ou externe à la bibliothèque, avait de ce fait un accès direct aux membres de la structure, en lecture et en écriture.

L'inconvénient de la définition d'un second format de transition est que l'utilisateur ne connaît plus *a priori* la structure d'une transition. Il doit faire un test avant d'accéder à la valeur qui l'intéresse. Par exemple, si le FSM est un graphe, la valeur du symbole d'entrée doit être lue dans le membre `stateN`, alors qu'elle sera lue dans le membre `symbol1` dans le cas d'une machine classique.

Pour éviter ces soucis à l'utilisateur, et donc diminuer les erreurs de manipulation, nous avons encapsulé l'accès aux transitions dans un objet, appelé `TransFnd` [9], dont la structure, inconnue de l'utilisateur, est la suivante :

```
(Type, Trans*, GraphTrans*)
```

[9] *Found Transitions*, transitions trouvées.

Au travers de cet objet, l'utilisateur pense manipuler des transitions classiques, de la forme `(symbol1, symbol2, weight, stateN)`. En réalité, ce sont les méthodes définies sur l'objet qui simulent ce format : en interne, elles testent le membre `Type` à la place de l'utilisateur, et vont lire ou écrire la valeur désirée dans le membre adéquat de la bonne structure (`Trans` ou `GraphTrans`).

Cet objet donne donc l'impression à l'utilisateur de manipuler des transducteurs pondérés, même dans le cas de graphes. Les méthodes définies sur cet objet sont :

1. Constructeur et destructeur :
   - `TransFnd_Create` : création de l'objet.
   - `TransFnd_Destroy` : destruction de l'objet.
2. Accès aux valeurs :
   - `TransFnd_GetNbTrans` : lecture du nombre de transitions.
   - `TransFnd_GetIn` : lecture du symbole d'entrée.
   - `TransFnd_GetOut` : lecture du symbole de sortie.
   - `TransFnd_GetWeight` : lecture du poids.
   - `TransFnd_GetState` : lecture de l'état suivant.
3. Modification des valeurs :
   - `TransFnd_SetIn` : écriture du symbole d'entrée.
   - `TransFnd_SetOut` : écriture du symbole de sortie.
   - `TransFnd_SetWeight` : écriture du poids.
   - `TransFnd_SetState` : écriture de l'état suivant.

Pour permettre l'accès aux transitions d'un état, l'objet `TransFnd` doit être initialisé : l'utilisateur demande à obtenir les transitions d'un état avant de les manipuler. Le Pseudocode 25 en donne un exemple.

Require: $pFSM$, un objet FSM, et $stateC$, l'indice d'un état
1: $pTransFnd \leftarrow$ `TransFnd_Create`()
2: `FSM_GetAllTrans`($pFSM, stateC, pTransFnd$)
3: $NbTrans \leftarrow$ `TransFnd_GetNbTrans`($pTransFnd$)
4: **for** $i \leftarrow 0$ to $i < NbTrans$ **do**
5: **if** `TransFnd_GetIn`$(pTransFnd, i) \neq$ `EPSILON` **then**
6: `TransFnd_SetOut`$(pTransFnd, i,$ `EPSILON`$) = 0$
7: **end if**
8: **end for**

Pseudocode 25: Exemple de manipulation des transitions

Exemple d'utilisation. Dans le cadre de la synthèse par sélection d'unités (cf. Section 10.2.5.4), une de nos machines a été pensée comme un graphe. La machine, construite sur plus de 39 000 symboles (des unités de parole), prend 159 Mo avant et après simplifications. Par contre, représentée sous la forme d'un graphe, elle ne prend plus que 79 Mo.

6.1.9 Synthèse

Implémentée en C et définie sur le semi-anneau tropical, notre bibliothèque met en œuvre les concepts de la programmation orientée objet : elle permet à l'utilisateur de manipuler des automates et des transducteurs, pondérés et non pondérés, au travers d'une interface commune où toute machine apparaît comme un transducteur pondéré. Dans ce contexte, un automate non pondéré est considéré comme un transducteur identitaire, dont les transitions et les états présentent systématiquement des poids nuls.

La bibliothèque propose la plupart des algorithmes standard de la littérature dans leur version pondérée, et utilise une représentation « point fixe » des pondérations, de manière à assurer la portabilité des applications développées. Elle autorise la création et la sauvegarde de machines dynamiques, et permet la sauvegarde et le chargement de machines binaires, plus compactes et plus rapides à charger.

Le développement de cette bibliothèque a été l'occasion de proposer des extensions aux représentations classiques des machines à états finis.

Les classes de symboles permettent de réduire, de manière significative, le nombre de transitions des machines dont le langage est de la forme $\Sigma^* \alpha$.

Les graphes binaires permettent de diviser par deux la taille des transitions, en évitant l'expression d'informations superflues, lorsque le langage a été pensé comme un graphe.

Ces extensions visent donc à réduire la taille des machines, dans des conditions bien précises. Ceci permet la conception de langages d'envergure, dans le cadre d'applications réelles.

6.2 Le compilateur Ovide

Développé sur la base de la nouvelle bibliothèque de machines à états finis, Ovide est un compilateur de règles de réécriture pondérées, de langages réguliers et de dictionnaires sous la forme de machines à états finis.

Le nom du compilateur est une référence à l'auteur latin Ovide [10] qui composa *Les Métamorphoses*, recueil de poèmes dont le héros se transforme toujours, tôt ou tard, en un être ou une chose dont il partage les qualités. Arachné par exemple, jeune fille qui excelle dans l'art du *tissage*, sera transformée en *araignée* par Athéna...

[10] Publius Ovidius Naso (43 ACN–17 PCN).

Le compilateur, quant à lui, convertit des relations régulières ou des langages réguliers en machines à états finis équivalentes.

Nous commençons par exposer les principes qui régissent la compilation des relations, des langages et des dictionnaires. Nous présentons ensuite les facilités offertes par le compilateur : la possibilité d'utiliser des alphabets standard ou définis par l'utilisateur, la déclaration de classes et le mécanisme d'inclusion de fichiers. Un point suivant décrit les différents modes de compilation autorisés. Enfin, nous présentons une extension des règles de réécriture, qui facilite la conception de règles complexes, sans risque d'erreurs de modélisation : les marqueurs. Cette section se referme sur une synthèse des caractéristiques principales du compilateur.

Note 6.2.1. L'Annexe B contient la documentation de notre compilateur, qui présente dans le détail la syntaxe à respecter dans les fichiers de règles. De ce fait, nous nous concentrons ici sur les *concepts* et les *principes* qui régissent le compilateur, ne nous référant à la syntaxe que lorsque ceci est nécessaire à la clarté du propos.

6.2.1 Principes

Ovide a été développé avant tout afin de compiler des règles de réécriture pondérées. Dans ce contexte et pour un alphabet Σ donné, le langage par défaut sur lequel s'appliquent les règles est le monoïde libre Σ^*. Cependant, il arrive qu'une application ait besoin d'un langage plus restreint, raison pour laquelle nous autorisons la définition d'un langage régulier ou d'un dictionnaire.

Ovide peut cependant être utilisé afin de compiler un simple langage régulier ou un dictionnaire, sans définition de règles de réécriture.

Règles de réécriture. Une section *Règle* permet de définir une liste de règles de réécriture. L'algorithme implémenté est celui de Mohri & Sproat (cf. Section 5.3), qui autorise l'expression de règles pondérées. Ovide compile chaque règle séparément, et les compose ensuite dans l'ordre proposé par le fichier, afin d'obtenir un seul transducteur représentant la totalité des règles.

Langages réguliers. Une section *Langage* accepte la définition de langages réguliers : chaque ligne de la section contient une expression régulière compilée sous la forme d'un automate selon l'algorithme de McNaughton et Yamada (cf. Section 5.2). Les différentes lignes de la section sont considérées comme des *alternatives*. Par exemple,

$$ab^* \qquad (6.2.1.1)$$
$$(c|d)e$$

est interprété comme

$$ab^* \mid (c|d)e \qquad (6.2.1.2)$$

Le langage dans sa globalité correspond donc à l'union des automates correspondants aux différentes lignes de la section.

Dictionnaires. Dans une section *Dictionnaire*, chaque ligne contient une expression combinant uniquement des symboles de l'alphabet à l'aide de l'opérateur de concaténation [11]. L'avantage de cette section est qu'il n'est pas nécessaire de compiler séparément les lignes avant de les réunir en un seul automate. Au contraire, un seul automate peut être directement construit.

La Figure 6.5 illustre le processus de création d'un dictionnaire. Le principe est de créer un automate avec le premier mot du lexique, et d'y *insérer* les autres mots. Pour ce faire, l'algorithme essaie de parcourir le mot à ajouter à partir de l'état initial de l'automate existant. Si le mot est complètement parcouru, l'état atteint devient final (s'il ne l'était pas). Par contre, si une partie du mot ne peut être parcourue, de nouveaux états qui lui sont dédiés sont créés *à partir de l'état atteint*. En somme, la construction d'un dictionnaire crée un automate déterministe de forme arborescente. Lorsque tous les mots ont été ajoutés à l'automate, la machine est minimisée.

Dans la Figure 6.5, les mots ajoutés sont dans l'ordre *couvent*, *couvera*, *coulent* et *cou*.

6.2.2 Facilités

Dans un fichier « Ovide », il est possible d'utiliser différents alphabets, de définir des classes et d'inclure d'autres fichiers « Ovide ».

6.2.2.1 Les alphabets

Le principe de base est que l'utilisateur déclare un alphabet d'entrée, `alphain`, et un alphabet de sortie, `alphaout`. Cependant, si les règles travaillent sur un seul alphabet ou que le fichier décrit simplement un langage ou un dictionnaire, l'utilisateur peut déclarer uniquement l'alphabet d'entrée.

Le compilateur propose quelques alphabets standard (au choix : alphabet `ASCII` étendu, caractères alphabétiques, caractères numériques ou caractères alphanumériques), mais permet à l'utilisateur de déclarer son propre alphabet, décrit dans un fichier respectant une syntaxe donnée.

La distinction entre `alphain` et `alphaout` implique qu'Ovide puisse proposer une distinction équivalente au niveau du langage ou du dictionnaire. C'est ainsi qu'Ovide propose les sections :

- `LANGIN` et `LANGOUT` pour les langages,
- `DICIN` et `DICOUT` pour les dictionnaires.

[11] Cet opérateur n'étant pas exprimé, une ligne du dictionnaire est une simple suite de symboles.

Bien sûr, les sections définies sur l'alphabet d'entrée n'acceptent pas de symboles appartenant à l'alphabet de sortie, et inversement.

Si deux alphabets sont définis, la section des règles utilise les alphabets comme suit. Pour une règle $\phi \rightarrow \psi :: \lambda\_\rho$, la partie supérieure de la règle, $\lambda\,\phi\,\rho$, est définie sur `alphain` et seule la réécriture ψ est définie sur `alphaout` [12].

Identification numérique. Dans Ovide, il est possible de référencer un symbole à l'aide de son numéro d'ordre dans l'alphabet. Le symbole peut être identifié soit en notation décimale, soit en notation hexadécimale. Par exemple, les deux notations suivantes identifient le dixième symbole de l'alphabet :

```
\10
\x0A
```

Ceci permet à l'utilisateur de faire référence aisément à des symboles difficiles à représenter, comme, par exemple, les 32 premiers symboles de l'alphabet `ASCII`.

Note 6.2.2. Il est important de mentionner que notre bibliothèque et notre compilateur commencent la numérotation des symboles à 1. Le 0 est réservé à la représentation de ϵ. Dans la suite de ce document, lorsque nous proposons des exemples de règles au format Ovide, ϵ est toujours noté au format hexadécimal :

```
\x00
```

6.2.2.2 Les classes

Afin de faciliter l'écriture des règles et des langages, des classes peuvent être définies.

Définition 6.2.1 (classe)**.** *Dans Ovide, une classe est un identifiant unique employé à la place d'une expression régulière.*

La définition d'une classe comprend deux parties : l'identifiant de la classe et l'expression régulière correspondante. L'identifiant de la classe est une suite ininterrompue de un ou plusieurs caractères alphanumériques. Il est séparé de l'expression par au moins un espace. Par exemple,

```
WORD    [a-z]*
```

[12] Ceci suppose une petite manipulation en interne. Tant que toutes les règles n'ont pas été composées ensemble, chaque règle travaille sur un alphabet interne qui est l'union d'`alphain` et d'`alphaout`. Dans cet alphabet interne exclusivement numérique, les symboles d'`alphaout` sont incrémentés du nombre de symboles d'`alphain`, afin d'éviter toute confusion entre les deux alphabets.

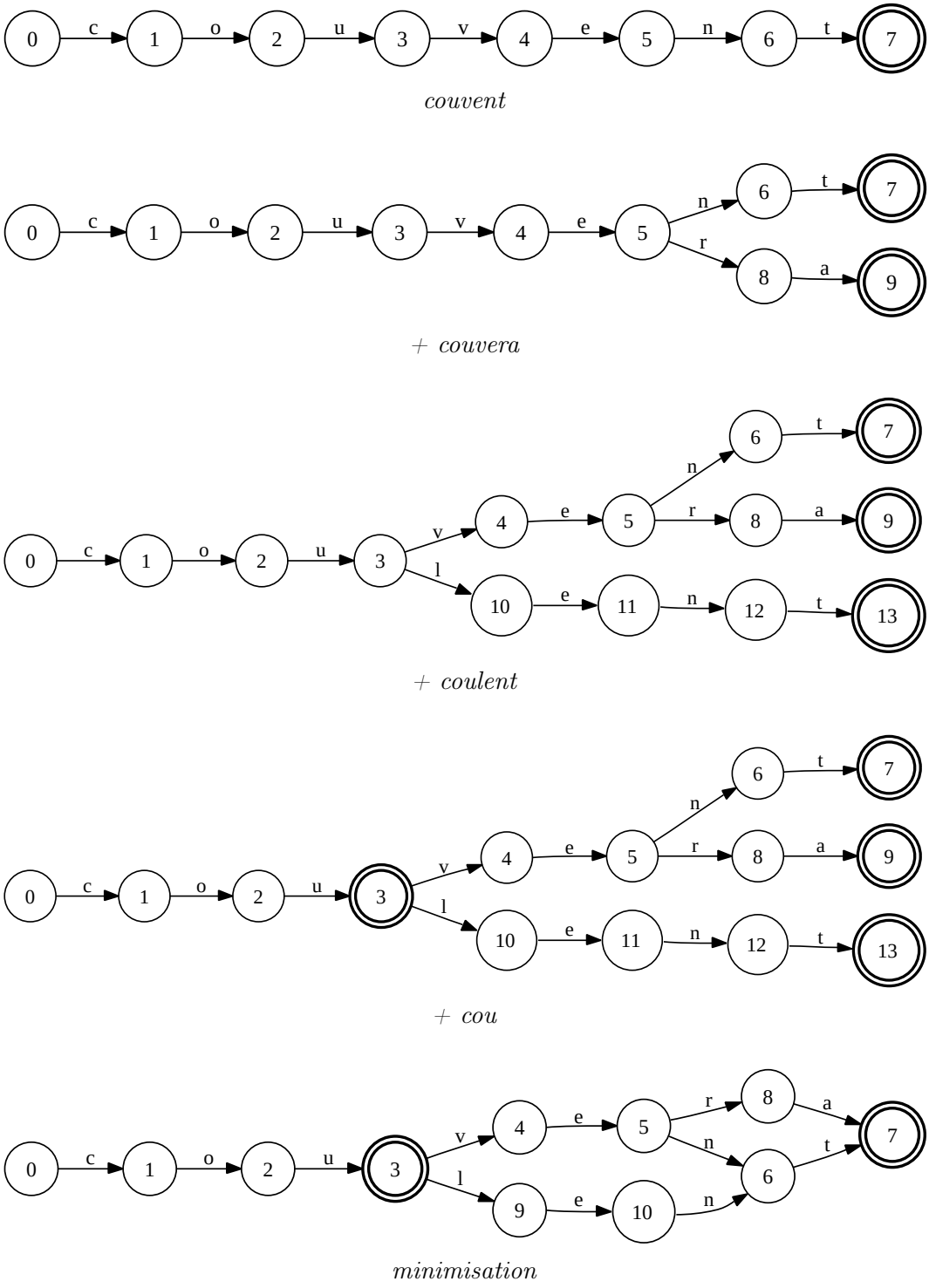


FIG. 6.5: Ovide – Construction d'un dictionnaire

définit la classe `WORD`, correspondant à l'expression régulière `[a-z]*`. Dans les sections autorisées à employer des classes, l'appel d'une classe se fait entre équerres :

```
<WORD>
```

Il est possible de définir des classes sur `alphain` dans une section `CLASSIN`, et des classes sur `alphaout` dans une section `CLASSOUT`. Les classes définies sur `alphain` peuvent être employées dans `LANGIN` et dans la partie supérieure des règles de réécriture ($\lambda\,\phi\,\rho$). Les classes définies sur `alphaout` peuvent être employées dans `LANGOUT` et dans la réécriture des règles (ψ). Les dictionnaires `DICIN` et `DICOUT`, par définition, ne peuvent recourir aux classes.

6.2.2.3 Inclusion de fichiers

Ovide autorise un mécanisme d'inclusion de fichiers, dont l'objectif est de modéliser le principe de la composition en cascade : chaque fichier décrit une relation, un langage ou un dictionnaire, et les différents fichiers sont composés ensemble afin de créer des relations complexes à partir de relations plus simples.

Dans Ovide, un même fichier peut en inclure plusieurs autres, et un fichier inclus peut lui-même en inclure d'autres. Ce principe implique que deux fichiers qui doivent être composés ensemble aient une interface *compatible* : l'alphabet sur lequel la composition des fichiers est calculée doit être *identique*. Etant donné trois alphabets distincts `A`, `B` et `C`, et sachant que la notation `{X:Y}` décrit un fichier défini sur les alphabets `X` et `Y`, le mécanisme d'inclusion autorise dès lors :

$$\begin{cases} \texttt{\{A:A\}} \circ \texttt{\{A:(A|B)\}} \\ \\ \texttt{\{A:B\}} \circ \texttt{\{B:(A|B|C)\}} \end{cases}$$

Comme nous l'expliquons ci-dessous dans le point relatif à la compilation, un appel explicite à un fichier inclus peut être nécessaire. Si un fichier « `Abc` » est inclus, l'appel du fichier se fait en précédant le nom du fichier par l'arobas :

```
@Abc
```

6.2.3 Sections et compilation

Compilation par défaut. Le mécanisme de compilation par défaut d'Ovide accepte les sections suivantes :

1. `INFO` : lieu d'informations générales, comme la définition des alphabets
2. `CLASSIN` : classes sur l'alphabet d'entrée
3. `CLASSOUT` : classes sur l'alphabet de sortie
4. `LANGIN` | `DICIN` : restriction sur le langage d'entrée

5. `RULE` : les règles de réécriture, pondérées ou non
6. `LANGOUT` | `DICOUT` : restriction sur le langage de sortie
7. `INCLUDE` : les fichiers inclus

Pour autant que toutes les sections soient définies, la compilation par défaut correspond à la cascade de compositions suivante :

$$(\texttt{LANGIN} \mid \texttt{DICIN}) \circ \texttt{RULE} \circ (\texttt{LANGOUT} \mid \texttt{DICOUT}) \circ \texttt{INCLUDE}$$

Compilation définie par l'utilisateur. La définition d'une section `COMPILE` interdit la compilation par défaut. Dans cette section, l'utilisateur peut utiliser des symboles de l'alphabet d'entrée, des classes définies sur l'alphabet d'entrée et des fichiers de la section `INCLUDE`, pour autant que les alphabets employés soient *compatibles*. Un fichier comportant une section `COMPILE` accepte donc les sections suivantes :

1. `INFO`
2. `CLASSIN`
3. `INCLUDE`
4. `COMPILE`

Dans la section `COMPILE`, tous les opérateurs définis sur les expressions régulières peuvent être utilisés (union, complément...), ainsi que la composition et la projection.

Par exemple, si l'alphabet d'entrée est `ASCII`, que la classe `CHAR` ([`a-z`]) est définie, et que le fichier `Ascii2Pho` (convertissant des graphèmes `ASCII` en phonèmes) est inclus, l'utilisateur peut définir une ligne de compilation telle que :

```
(abc<CHAR>* ∘ @Ascii2Pho)>o
```

qui se lit comme ceci : la string `abc` suivie de 0 à n caractères [`a-z`], est composée avec la relation décrite dans le fichier `Ascii2Pho`. Du résultat obtenu, on ne garde que la seconde projection (`>o`).

6.2.4 Les marqueurs : extension des règles de réécriture

6.2.4.1 Positionnement du problème

Dans le principe, les règles de réécriture sont relativement faciles à concevoir : il *suffit* de déterminer le contexte d'application dans lequel une réécriture peut être appliquée.

Construire un ensemble de règles, cohérent et sans erreur de modélisation, est cependant un exercice délicat, et ce pour plusieurs raisons :

1. Un symbole réécrit ne se distingue en rien d'un symbole original. Or,
   - Le concepteur désire parfois que les règles ne s'appliquent qu'à des symboles originaux. Posons les deux règles suivantes :

     (1) $a \rightarrow b :: c \_ d$
     (2) $b \rightarrow e :: c \_$

     L'application successive de ces règles convertira la string *cad* en *ced*. Le résultat n'est donc pas atteint, puisque l'application de la deuxième règle ne doit concerner qu'un b original. Dans le cas des règles suivantes :

     (1) $a \rightarrow \epsilon :: c \_ d$
     (2) $c \rightarrow e :: \_ d$

     L'application successive de ces règles convertira la string *cad* en *ed*. Or, la deuxième règle ne peut s'appliquer que lorsque c est dans le contexte de d dans la string originale.
   - A l'inverse, certaines règles ne doivent s'appliquer qu'à des symboles réécrits ou dans le contexte de symboles réécrits. Or, ces règles s'appliqueront également aux symboles originaux.
2. Une règle peut parfois s'appliquer en différents lieux d'une séquence, pour autant qu'une condition relative à l'ensemble de la séquence soit respectée. En voici un exemple :« *un seul* b *peut être inséré en n'importe quel point d'une string de 5 caractères* ». Cette simple règle nécessitera 6 règles de réécriture pour être complètement exprimée :

   (1) $\epsilon \rightarrow b :: \hat{} \_.\{5\} \$$
   (2) $\epsilon \rightarrow b :: \hat{} .\{1\} \_.\{4\} \$$
   (3) $\epsilon \rightarrow b :: \hat{} .\{2\} \_.\{3\} \$$
   (4) $\epsilon \rightarrow b :: \hat{} .\{3\} \_.\{2\} \$$
   (5) $\epsilon \rightarrow b :: \hat{} .\{4\} \_.\{1\} \$$
   (6) $\epsilon \rightarrow b :: \hat{} .\{5\} \_ \$$

   Cette répétition peut être la cause de certains oublis.

Afin de proposer une solution élégante à ces difficultés de modélisation, nous avons défini la notion générale de *marqueur*.

Note 6.2.3. Cette notion est fortement inspirée de la notion de *gène marqueur* définie en biologie (Miki & Mchugh 2004) : un gène marqueur est un gène inséré dans une construction génétique, de manière à dépister plus facilement les transformations réalisées. Le gène marqueur, pour être identifiable, doit être étranger au génome de l'organisme modifié.

6.2.4.2 Définitions

Définition 6.2.2 (Marqueur)**.** *Un* marqueur *est un symbole, extérieur à l'alphabet utilisé, que l'on insère dans une règle de réécriture afin d'identifier un phénomène et d'en suivre l'évolution.*

Nous avons défini trois types de marqueurs : le *déclencheur*, le *masqueur* et le *bloqueur*.

Définition 6.2.3 (Déclencheur)**.** *Le* déclencheur *est un marqueur qui permet de signaler de manière non ambiguë qu'une condition d'application a été rencontrée, de manière à déclencher, en toute sécurité, l'application d'une ou de plusieurs règles de réécriture.*

Le déclencheur δ est donc inséré par une règle, qui constate qu'une condition est remplie. D'autres règles peuvent ensuite préciser δ, afin de conditionner leur application à sa présence. Dans l'exemple suivant,

(1) $\epsilon \rightarrow \delta :: [condition]$
(2) $a \rightarrow b :: \_\, \delta\, def$
(3) $f\, \delta \rightarrow g :: \_\, def$
(4) $\delta \rightarrow \epsilon$

la première règle insère le déclencheur. La seconde s'applique en présence du déclencheur, mais ne le réécrit pas. Ceci signifie qu'une autre règle, qui préciserait le même déclencheur, pourrait également s'appliquer. La troisième règle, par contre, réécrit le déclencheur, ce qui empêche l'application d'autres règles par la suite. La quatrième règle supprime le déclencheur éventuellement encore présent, et devenu inutile.

Définition 6.2.4 (Masqueur)**.** *Le* masqueur *est un marqueur qui prend temporairement la place d'une expression régulière, afin d'éviter qu'une règle de réécriture ne s'applique à tort sur cette expression.*

Typiquement, un masqueur μ est utilisé à la place de la réécriture (ψ) de la règle. En somme, la cible de la règle est réécrite par un masqueur μ attribué à une réécriture particulière. Il est donc nécessaire qu'une autre règle, située à la fin de l'ensemble de règles, convertisse μ en la réécriture qu'il masque. En voici un exemple :

(1) $a \rightarrow \mu :: c \_\, d$
(2) $b \rightarrow e :: c \_\, d$
(3) $\mu \rightarrow b$

Dans l'exemple, la première règle masque la réécriture b à l'aide du masqueur μ. Ceci évite à la deuxième règle, dont la cible est b et qui concerne le même contexte que la première règle, de pouvoir s'appliquer sur une entrée qui aurait été réécrite par la première règle. La troisième règle peut enfin, en toute sécurité, convertir μ en b, sans risque de réécriture erronée.

Définition 6.2.5 (Bloqueur)**.** *Le* bloqueur *est un marqueur qui s'insère temporairement entre deux expressions régulières, afin d'éviter la formation d'une nouvelle expression régulière sur laquelle une règle de réécriture pourrait s'appliquer à tort.*

Le bloqueur est par exemple utile dans le cas d'une règle qui supprime une cible, étant donné que cette suppression peut entraîner la formation d'un contexte favorable à l'application inappropriée d'une autre règle. Comme le masqueur, le bloqueur β doit être réécrit, lorsque tout risque de réécriture abusive a disparu :

(1) $a \rightarrow \beta :: c \_ d$
(2) $c \rightarrow e :: \_ d$
(3) $\beta \rightarrow \epsilon$

6.2.4.3 Intégration des marqueurs dans Ovide

Dans Ovide, un marqueur se définit dans la section des classes. Le marqueur se distingue cependant d'une classe, parce que l'expression régulière autorisée se limite à l'*esperluette* (`&`), suivie d'un entier (`\d+`). Par exemple :

```
BDECL    &1
BMASK    &2
BBLOK    &3
```

C'est l'expression qui distingue donc le marqueur de la classe. La syntaxe particulière du marqueur assure ainsi que le compilateur l'identifie sans ambiguïté, et évite de le confondre avec une séquence de symboles de l'alphabet : en interne, le marqueur se voit automatiquement attribuer une valeur numérique, qui correspond à la valeur de la déclaration, à laquelle s'additionne le nombre de symboles de l'alphabet. Un marqueur ne supprime donc aucune séquence de symboles du langage modélisé.

Le marqueur peut être employé dans les mêmes sections que la classe : dans la définition d'un langage ou dans une règle de réécriture. L'appel du marqueur est identique à l'appel d'une classe, réalisé entre équerres :

```
<BDECL>
```

On constate que la définition d'un marqueur ne spécifie en aucune manière son type. C'est l'usage que l'utilisateur en fait qui détermine le type du marqueur. Le marqueur `BDECL`, par exemple, aura la valeur d'un déclencheur dans les règles suivantes, parce que l'usage qui en est fait est celui d'un déclencheur :

```
\x00 → <BDECL> :: _ ·{1,5}
a <BDECL> → b :: c _ d
```

A la fin de la compilation d'un fichier de règles, il est possible que des marqueurs soient encore présents dans le FSM correspondant. Ceci est autorisé, afin de permettre la

segmentation d'un modèle entre plusieurs fichiers de règles. Un marqueur peut donc être introduit par un fichier, mais utilisé et supprimé par un autre. La plupart des fichiers de règles que nous avons définis dans le cadre de la correction orthographique (cf. Partie III, Chapitre 15) exploitent cette facilité offerte par Ovide.

Astuce. La définition des marqueurs ne semble pas résoudre le problème des règles pouvant s'appliquer en différents lieux d'une séquence. L'exemple que nous avions donné était : « *un seul* b *peut être inséré en n'importe quel point d'une string de 5 caractères* ». La solution, dans Ovide, vient de la combinaison des marqueurs et de différentes sections :

```
[CLASSIN]
  L5    &1
  INS   &2
[RULE]
  \x00 → <L5> :: ^ _ .{5} $
  \x00 → <INS>
[LANGOUT]
  <L5> [^<INS>]* <INS>? [^<INS>]*
```

Dans la section [CLASSIN], deux déclencheurs sont créés. Dans la section [RULE], une première règle utilise L5 pour marquer les séquences de 5 caractères. Une seconde règle utilise INS pour indiquer un point de la séquence où une insertion est réalisée. La règle, telle qu'elle est écrite, autorise une insertion en n'importe quel point de n'importe quelle séquence. C'est le langage décrit dans [LANGOUT] qui contraint les insertions acceptées : ce langage indique que toute séquence de 5 caractères ne peut accepter qu'une seule insertion, mais en n'importe quel point.

L'insertion d'un *b* en particulier peut être réalisée, en toute sécurité, dans un autre fichier Ovide. La règle sera dès lors simplement :

```
<INS> → b
```

Cette modélisation en deux fichiers est très pratique, parce qu'elle permet de concevoir les contraintes sur le langage et les contraintes sur les réécritures de manière séparée. La tâche en est fortement facilitée. Ce principe nous a par exemple permis de modéliser des distances d'édition complexes dans le cadre de la correction orthographique (cf. Partie III, Chapitre 15).

6.2.5 Synthèse

Ovide est avant tout un compilateur de règles de réécriture pondérées, qui utilise un mécanisme d'inclusion de fichiers afin d'autoriser la conception de relations complexes

à partir de relations plus simples, modélisables par un spécialiste du domaine d'application ou *via* un apprentissage [13].

Ovide permet également la description de langages réguliers et de dictionnaires, qui peuvent être employés seuls ou en combinaison avec des règles de réécriture.

Si la syntaxe des règles de réécriture est bien définie, la portée des règles est parfois difficile à contrôler, ce qui entraîne des erreurs de modélisation. Pour résoudre ce problème, nous avons proposé une extension aux règles de réécriture : les marqueurs. A l'aide de marqueurs, il est possible d'indiquer, de manière non ambiguë, le contexte d'application des règles construites. En combinaison avec le mécanisme d'inclusion de fichiers, les marqueurs permettent en outre à l'utilisateur de concevoir, dans des réflexions séparées, les contraintes relatives aux langages de celles relatives aux contextes d'application.

Ovide a été conçu autour de notre nouvelle bibliothèque de machines à états finis. Le résultat de la compilation d'un fichier est de ce fait une machine à états finis et un ou deux alphabets, sauvegardés au format binaire. Dans le cadre d'une application, ces fichiers binaires peuvent être chargés et utilisés à l'aide des méthodes de notre bibliothèque.

Les machines et les alphabets utilisés dans le cadre de nos applications ont été modélisés à l'aide d'Ovide. Des exemples de fichiers Ovide et des résultats de la compilation illustrent la présentation des différents modèles proposés.

[13] Dans ce cas, le fichier « Ovide » peut être généré automatiquement par un langage de script, comme Perl.

Chapitre 7

Conclusion

Les automates et les transducteurs, vus comme des graphes orientés étiquetés, proposent différents niveaux de représentation. Certains, comme les machines présentant des transitions ϵ, sont le résultat d'opérations plus ou moins complexes. D'autres, comme les machines déterministes et minimales, sont le résultats d'algorithmes d'optimisation, qui assurent une représentation de la machine en un minimum de place, et un parcours d'une string d'entrée linéairement proportionnel à la taille de la string. Ces machines, équivalentes aux langages réguliers, sont closes sous les opérations régulières telles que l'union, la concaténation ou l'étoile de Kleene. Les transducteurs, pour leur part, sont également clos sous la composition, opération d'une importance capitale en traitement des langues, parce qu'elle permet de modéliser des relations complexes à partir de relations beaucoup plus simples.

Si l'intérêt des chercheurs pour ces outils a été remis en cause dans le domaine du traitement du langage naturel, parce que les langages réguliers n'autorisent pas les imbrications infinies, les extensions proposées aux algorithmes classiques définis sur les automates et les transducteurs, et applicables aux automates et transducteurs pondérés, permettent de gérer un certain degré d'incertitude. La définition, entre autres, d'un algorithme de recherche des n meilleurs chemins d'un graphe pondéré a ouvert la voie à une nouvelle technologie du langage, basée sur des machines à états finis pondérées, là où auparavant *modèles de langue* rimaient inéluctablement avec *programmation dynamique* ou *modèles de Markov cachés*. Ces machines pondérées sont la base des applications que nous présentons en Parties II et III.

Complétées par les outils de description syntaxique que sont les expressions régulières et les règles de réécriture, les machines à états finis forment une « boîte à outils » dont la seule limite, peut-être, est de ne pas autoriser les imbrications infinies.

Sur la base de ce constat et en l'absence d'outils répondant aux besoins de nos applications, nous avons développé notre propre bibliothèque et notre propre compilateur d'expressions régulières et de règles de réécriture pondérées.

Notre bibliothèque, développée en C ANSI, mais basée sur les concepts de la programmation orientée objet, propose une représentation unique des automates et des trans-

ducteurs, pondérés et non pondérés, dont les poids ont une représentation « point fixe », afin d'assurer la portabilité de la bibliothèque sur plateforme embarquée. Le développement de cette bibliothèque a été l'occasion de proposer deux extensions aux représentations classiques des machines à états finis. La première est la définition de classes de symboles, qui condensent en une seule transition l'ensemble des transitions, de même poids, qui relient deux états d'une machine. La seconde est la définition de graphes orientés pondérés, qui font l'économie des symboles qui étiquettent classiquement les transitions d'une machine, lorsque le langage représenté a été pensé et conçu comme un graphe, dont les états sont les symboles de l'alphabet. Ces deux extensions réduisent, de manière significative, la taille des machines compilées avec notre bibliothèque.

Notre compilateur permet, quant à lui, la description de règles de réécriture, de langages et de dictionnaires, et autorise la modélisation de relations complexes à partir de relations simples, grâce à un mécanisme d'inclusion récursif. Dans le cadre du développement de ce compilateur, nous avons proposé une extension aux règles de réécriture : les marqueurs, qui identifient de manière non ambiguë le contexte d'application d'une règle, facilitent l'expression de contraintes et évitent les erreurs de modélisation. La combinaison des marqueurs et du mécanisme d'inclusion a fortement facilité l'expression des modèles que nous présentons dans la suite de ce document.

Deuxième partie

Synthèse par sélection d'unités non uniformes

Chapitre 8

Introduction

La synthèse de la parole est le processus qui consiste à produire de la parole à partir du texte. Il s'agit donc de générer un *signal acoustique* de parole à partir d'une *représentation symbolique*, le texte. Un système de synthèse à partir du texte est classiquement divisé en deux modules distincts : un module de traitement automatique de la langue écrite, qui produit une représentation symbolique non ambiguë du texte, et un module de traitement du signal, qui produit le signal de parole à partir de cette représentation non ambiguë.

Cette partie se concentre sur le module de traitement du signal, et plus particulièrement sur un sous-module du traitement du signal : le module de sélection d'unités de parole, apparu du fait de l'évolution du concept de synthèse de la parole. Elle manipule en outre de nombreuses notions linguistiques, sur lesquelles se fondent notre raisonnement et les hypothèses qui ont mené à la mise en place du système proposé.

8.1 Plan de la partie

Chapitre 8 : définition de quelques concepts fondamentaux, sur lesquels se fonde notre propos. Nous rappelons ce qu'est la parole en termes acoustiques. Nous décrivons ensuite le type de représentation linguistique non ambiguë utilisée classiquement en synthèse. Cette représentation est la base de toute réflexion ultérieure. Nous expliquons enfin en quoi le concept de synthèse a évolué pour devenir ce qu'il est actuellement, c'est-à-dire un processus régi par la sélection d'unités de parole non uniformes. Sur cette base, nous présentons les concepts généraux de la synthèse par sélection d'unités et nous en dégageons les critères qui déterminent la qualité de la synthèse obtenue. Parmi ces critères figure le module de sélection, auquel nous avons consacré toute notre attention.

Chapitre 9 : état de l'art en synthèse par sélection d'unités non uniformes. Ce chapitre permet de situer la portée de notre contribution, en proposant une analyse des principes et des limites des modules de sélection de l'état de l'art. Ces principes se répartissent entre critères de sélection, et méthodes d'optimisation.

Chapitre 10 : présentation de notre nouveau système de sélection, qui se nomme LiONS pour « *Linguistically Oriented Non-uniform units Selection* ». Nous détaillons d'abord les critères de sélection retenus, juste compromis entre informations linguistiques et acoustiques, et le processus de sélection initialement développé. Nous décrivons ensuite les réflexions et les étapes permettant d'optimiser ce nouveau système à l'aide de machines à états finis, et les révisions qui ont accompagné cette optimisation.

Chapitre 11 : conclusion de cette partie, dans laquelle nous faisons le point sur la portée de notre contribution au domaine, et sur l'intérêt général des modèles proposés.

8.2 Le signal acoustique de parole

Physiquement, le signal de parole est une variation de la pression de l'air due à l'activité de l'appareil phonatoire de l'être humain, dont la Figure 8.1 présente le schéma. Lors de la production de parole par l'appareil phonatoire, l'air est expulsé des poumons,

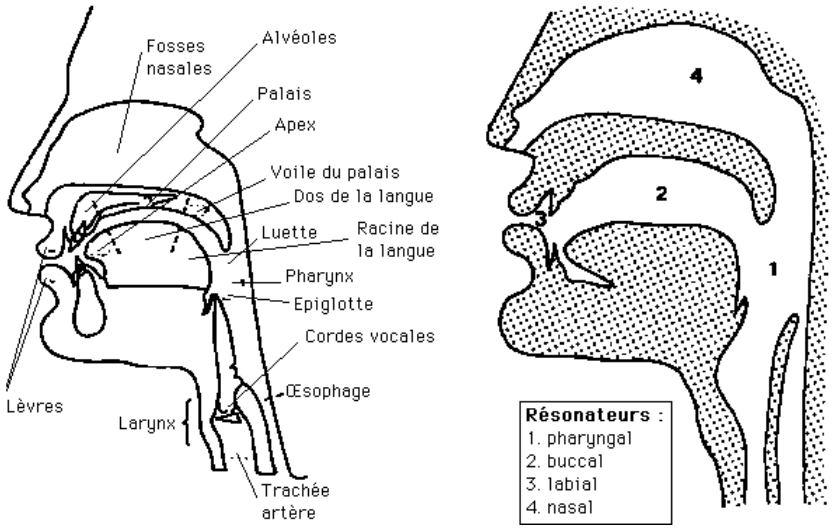


FIG. 8.1: Appareil phonatoire de l'être humain

remonte la trachée, traverse le larynx où il rencontre les cordes vocales qui vibrent ou ne vibrent pas, et passe par une série de résonateurs plus ou moins activés. Les résonateurs qui interviennent sont la zone pharyngale, la cavité buccale, les fosses nasales et la zone labiale. Une production de l'appareil phonatoire est donc le résultat de l'état des cordes vocales et de la forme du conduit vocal qui, ensemble, agissent comme un *filtre* sur le flux d'air provenant des poumons.

Au niveau acoustique, cette variation de la pression de l'air peut être décomposée en *traits acoustiques* dont les principaux sont le spectre, la fréquence fondamentale, les formants, l'énergie et la durée.

Notions fondamentales. Nous rappelons qu'un signal est dit *périodique* si son amplitude évolue selon un schéma qui se répète sur une période de temps constante T. La *fréquence* F d'un signal périodique est le nombre de fois que la période se reproduit en une seconde :

$$F = \frac{1}{T} \tag{8.2.0.1}$$

La fréquence se mesure en Hertz (Hz) et une fréquence de 1 période par seconde vaut 1 Hz. La Figure 8.2 donne un exemple de signal périodique, représenté dans l'espace temporel.

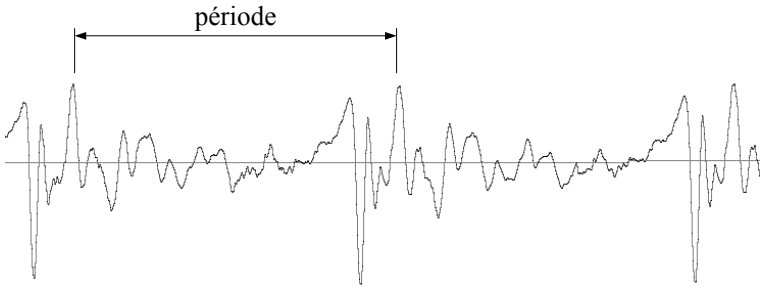


FIG. 8.2: Signal périodique

Lorsque l'on convertit un signal de l'analogique au numérique, on prélève sur le signal des échantillons à intervalles réguliers. C'est ce que l'on appelle la *fréquence d'échantillonnage*. Selon le théorème de Nyquist-Shannon, pour obtenir une conversion correcte du signal analysé, la fréquence d'échantillonnage F_e doit être au moins égale à 2 fois la fréquence maximale F_{max} de ce signal :

$$F_e = \frac{1}{T_e} \geq 2F_{max} \tag{8.2.0.2}$$

Sans entrer dans le détail, ce théorème évite d'obtenir un repli spectral du signal de parole. La parole humaine ne dépassant pas les 8 KHz, une fréquence d'échantillonnage de 16KHz est suffisante.

La conversion numérique implique également de déterminer le nombre fini de valeurs discrètes selon lesquelles chaque échantillon sera représenté. Il s'agit de la *quantification*, qui se mesure en bits. Par exemple, une quantification de 16 bits pour une fréquence d'échantillonnage de 16 KHz permet d'échelonner les valeurs de 0 à 8 KHz sur 2^{16} (65 536) valeurs différentes.

Nous incitons le lecteur qui ne maîtriserait pas ces notions et désirerait se documenter à se reporter à (Boîte *et al.* 2000) pour de plus amples explications.

8.2.1 Le spectre

Le spectre d'un signal acoustique complexe comme la parole est l'ensemble des fréquences qui le constituent. L'évolution des fréquences présentes dans un signal acoustique au cours du temps peut être visualisée à l'aide d'un spectrogramme, comme celui de la Figure 8.3.

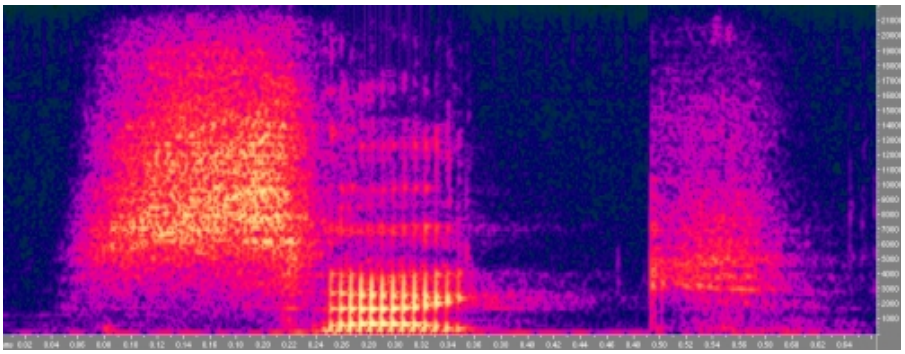


FIG. 8.3: Spectrogramme temps/fréquence. Le mot prononcé est « sac »

8.2.2 La fréquence fondamentale

La fréquence fondamentale, couramment abrégée *F0*, est la fréquence la plus basse du signal acoustique. Cette fréquence fixe la *hauteur* du signal : par exemple, en musique, un *la* a une fréquence fondamentale de 440 Hz. Dans un signal de parole, la fréquence fondamentale est générée par les vibrations des cordes vocales. Elle évolue dans le temps, en fonction du signal à produire et de l'état de l'appareil phonatoire. En un point donné, si le signal présente une fréquence fondamentale, le signal est voisé. Dans le cas contraire, il est non voisé. Les voyelles sont toujours voisées tandis que certaines consonnes sont non voisées. Nous revenons sur ces distinctions dans la section 8.3 de ce Chapitre.

Le signal contient également des *multiples* de la F0, les *harmoniques*. Le nombre d'harmoniques et leurs amplitudes peuvent varier, ce qui détermine le *timbre* du signal. En musique par exemple, c'est ce qui explique la différence perçue lors de la production d'une même note par des instruments différents, comme un piano et un violon. En parole, les harmoniques sont fonction de la tension et de l'épaisseur des cordes vocales, et déterminent la différence de timbre de voix entre deux locuteurs.

8.2.3 Les formants

Un formant est une zone de fréquences du spectre vocal qui est amplifiée par un ou plusieurs résonateurs de l'appareil phonatoire. Lors d'une production sonore, certaines

fréquences sont donc amplifiées si elles se situent dans la zone de fréquences amplifiée par les résonateurs mis en action.

Les formants sont présents dans toutes les productions sonores de l'appareil phonatoire, mais sont plus nets et plus marqués dans le cas des voyelles, dont ils déterminent le *timbre* : par exemple, comme le montre la Figure 8.4, un *a* se distingue d'un *i* par la position de ses formants, et non par sa fréquence fondamentale [1].

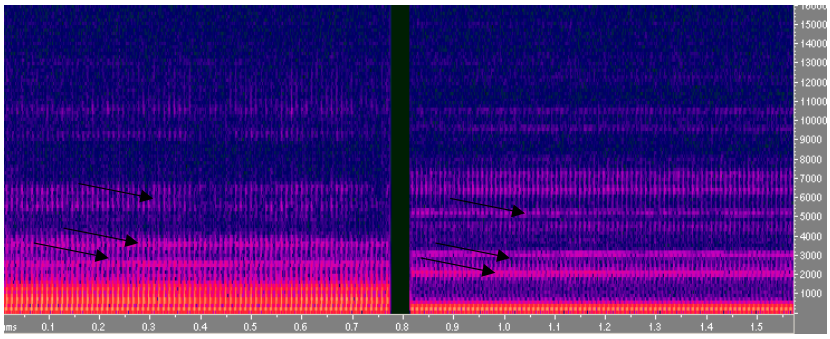


FIG. 8.4: Comparaison des trois premiers formants de *a* et *i*

8.2.4 L'énergie

Sans entrer dans des détails qui dépassent largement le propos de ce Chapitre, l'énergie du signal de parole est une fonction de l'évolution de l'*amplitude* du signal. L'énergie se calcule sur des portions de signal de 10 à 30 ms, pendant lesquelles le signal peut être considéré comme quasi stationnaire du fait de l'inertie des muscles de l'appareil phonatoire intervenant dans le processus de phonation. La mesure de l'énergie du signal permet entre autres de déterminer les zones du signal où l'estimation de la fréquence fondamentale a un sens. Elle permet également de détecter l'alternance entre parole et silence.

8.2.5 La durée

Dans le signal de parole, chaque son se produit pendant un certain temps. Ce trait acoustique est le résultat de certains facteurs, que nous ne faisons que citer parce qu'ils font appel à des notions présentées dans le point suivant de ce Chapitre :

1. Le souffle. La nécessité de reprendre son souffle implique de marquer des pauses régulières dans le flux de parole. Entre deux pauses, certaines portions du signal ont tendance à s'allonger.

[1] On remarque que le terme *timbre* fait référence en parole à deux réalités fort différentes : le timbre de la voix, déterminé par les harmoniques de la fréquence fondamentale, et le timbre des voyelles, déterminé par la position des formants.

2. La forme de l'appareil pour un son donné. Cette forme est plus ou moins propice à maintenir la prononciation du son. On comparera par exemple [s] et [p].
3. L'enchaînement des sons dans le flux de parole. En effet, la forme de l'appareil phonatoire avant et après un son donné influence la durée de ce son. On comparera par exemple le *o* de *mot* et de *rose*.

8.2.6 Quelques méthodes d'analyse et de représentation

8.2.6.1 Transformée de Fourier

Le passage d'une représentation temporelle à une représentation fréquentielle est réalisé à l'aide de la transformée de Fourier, définie par :

$$\mathtt{TF}[f(t)] = F(\omega) = \int_{-\infty}^{+\infty} f(t)\,\mathrm{e}^{-j\omega t}\,\mathrm{d}t \tag{8.2.6.1}$$

qui permet de décomposer un signal périodique complexe en une somme de signaux périodiques simples. Cette fonction permet donc d'obtenir le spectre du signal.

L'application de la transformée de Fourier s'appuie sur l'hypothèse que le signal est stationnaire (l'intégrale s'étend de $-\infty$ à $+\infty$). Or, comme nous l'avons mentionné, le signal de parole peut être considéré comme quasi stationnaire sur une période relativement courte de 10 à 30 ms. La transformée est donc généralement réalisée sur une fenêtre de 10 à 30 ms prise sur le signal.

Le passage d'une représentation fréquentielle à une représentation temporelle est réalisé à l'aide de la transformée de Fourier inverse, définie par :

$$\mathtt{TF}^{-1}[F(\omega)] = f(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(\omega)\,\mathrm{e}^{-j\omega t}\,\mathrm{d}\omega \tag{8.2.6.2}$$

8.2.6.2 Cepstre

Le cepstre [2] est une transformation d'un signal $x(n)$ du domaine temporel vers un autre domaine analogue au domaine temporel. La définition du cepstre la plus répandue en traitement de la parole est qu'il s'agit de la transformée de Fourier inverse, appliquée au logarithme de la transformée de Fourier du signal. :

$$\mathtt{TF}^{-1}(log_{10}(\mathtt{TF}(x(n)))) \tag{8.2.6.3}$$

L'intérêt du cepstre est qu'il permet de déconvoluer les contributions respectives de la source (bruit blanc ou F0) et du filtre (dont les formants).

[2] Prononcé [k ɛ p s t ʁ].

8.2.6.3 Coefficients LPC

Les coefficients LPC [3] sont les coefficients d'un modèle de prédiction qui permet de prédire un échantillon à partir des p échantillons précédents. p, qui correspond au nombre de coefficients du modèle, est appelé *l'ordre de la LPC*.

Le modèle permet aussi d'estimer l'enveloppe spectrale d'un signal de parole à l'instant t. A titre d'exemple, il faut 30 ms de signal de parole pour estimer 14 coefficients, à partir desquels il est possible d'estimer l'enveloppe spectrale. Une enveloppe de 256 points [4] est donc représentée par seulement 14 coefficients. Etant donné qu'une définition plus précise dépasserait largement le cadre de cette thèse, nous incitons le lecteur intéressé à se référer à (Markel & Gray 1980).

8.3 Représentation symbolique non ambiguë

Afin de générer un signal de parole compréhensible par l'être humain, le module de traitement du signal doit se comporter *de manière analogue* à l'appareil phonatoire de l'être humain, en faisant évoluer les traits acoustiques du signal selon le message à générer. Déterminer l'évolution de ces traits acoustiques demande de posséder des informations symboliques suffisantes et non ambiguës. C'est la raison pour laquelle, classiquement, la représentation symbolique non ambiguë d'un texte à des fins de synthèse est constituée de la séquence de *phonèmes* correspondant au texte, et de la description de la *courbe prosodique* à appliquer à cette séquence de phonèmes.

8.3.1 Le phonème

Etudié en phonologie, le phonème est la base de la parole. Unité abstraite, il se définit dans un *système d'oppositions* et se caractérise par des *traits articulatoires*. Bien qu'il s'agisse d'une unité abstraite, le phonème fait référence à un bruit ou à un son produit par l'appareil phonatoire de l'être humain (cf. Figure 8.1) et se réalise dans le discours au travers de ses *allophones*.

Note 8.3.1. Dans ce point, nous faisons une distinction entre *son* et *bruit*. Le son est un signal périodique, tandis que le bruit est non périodique. Lorsque nous faisons référence aux deux sans distinction, nous parlons de *production sonore*.

A titre d'exemple, la Table 8.1 présente les 37 phonèmes du français.

8.3.1.1 Le système phonologique

Définition 8.3.1 (Axe paradigmatique)**.** *Contrairement à l'axe syntagmatique qui organise les éléments d'un système le long de la ligne du temps, l'axe paradigmatique régit les*

[3] *Linear Prediction Coding.*
[4] Coefficients de Fourier.

Voyelles		**Consonnes**	
i	am***i***, f***i***lm	p	***p***as, dé***p***art, a***pp***orter
e	***é***pis, abb***é***	b	***b***as, dé***b***iter
ɛ	parl***ais***, pr***ê***t	m	***m***as, dé***m***uni
ɛ̃	dem***ain***, p***ein***ture		
a	m***a***lle, quitt***a***	f	***f***ais, dé***f***is, ***ph***ysique
		v	***v***ais, dé***v***ier
y	ém***u***, t***u***nique		
ø	p***eu***, v***eu***t	t	***t***as, a***tt***ention
œ	p***eu***ple, v***eu***lent	d	***d***é, a***d***apter
œ̃	***un***, empr***un***t	n	***n***on, é***n***umérer, bo***nn***e
ə	l***e***, d***e***mander		
		s	***s***as, a***ss***eoir
u	p***ou***le, ***ou***blier	z	***z***orro, arro***s***er
o	chev***au***x, v***ô***ter		
ɔ	p***o***rter, c***o***llision	ʃ	***ch***at, ***sch***éma
ɔ̃	b***on***té, c***om***pagn***on***	ʒ	***j***ouer, égor***g***er
ɑ	p***â***te		
ɑ̃	p***en***dre, ch***an***ter	ɲ	o***ign***on, moi***gn***on
		k	***c***as, a***cqu***iescer
Semi-voyelles		g	***g***ars, é***g***oïste
		ŋ	parki***ng***
j	pa***ille***, fi***lle***, p***i***ed		
w	***ou***i, r***ou***age	ʁ	***r***at, pa***r***tager
ɥ	h***u***it, h***u***ile	l	***l***ouer, va***ll***ée

TAB. 8.1: Phonèmes du français

possibilités de substitution des éléments d'un système en un point donné.

Définition 8.3.2 (Phonème)**.** *Dans le flux la parole, le phonème est l'unité minimale de distinction. Il se définit par opposition sur l'axe paradigmatique.*

Le phonème permet de distinguer ce que nous appelons des *images acoustiques*. Par exemple, [m i l] et [p i l] sont des images acoustiques distinctes : elles divergent par [m] et [p], qui sont de ce fait des phonèmes. Par nature, le phonème n'est pas porteur de sens, au contraire du morphème qui est lui l'unité minimale de *signification* (cf. Définition 13.1.2). En français par exemple, on trouve les morphèmes *chant-*, *boir-*, *-era*, *-ais*, *-iss-*, etc. En somme, le phonème permet de construire les images acoustiques des morphèmes, à partir

Aperture	**Lieu**						
	Palatal antérieur		central	Palatal postérieur		Vélaire	
1	i			y		u	
2	e			ø		o	
moyenne			ə				
3	ɛ	ɛ̃		œ	œ̃	ɔ	ɔ̃
4	a					ɑ	ɑ̃
Nasalité	oral	nasal	oral	oral	nasal	oral	nasal
Lèvres	étirées		-	arrondies			

TAB. 8.2: Description articulatoire des voyelles du français

desquels sont construits les mots de la langue.

8.3.1.2 Le phonème en termes articulatoires

En termes articulatoires, un phonème est une description abstraite de l'état des cordes vocales et de la forme du conduit vocal à l'aide de caractéristiques articulatoires. Les phonèmes se répartissent en trois classes : les *voyelles*, les *semi-voyelles* et les *consonnes*. Les caractéristiques articulatoires varient selon la classe considérée.

La voyelle. Une voyelle est un *son* : elle est le résultat de la vibration des cordes vocales sur l'air qui s'échappe de l'appareil phonatoire sans être freiné ni arrêté. Une voyelle se distingue d'une autre par son *timbre*, qui, en termes articulatoires, est le résultat de la combinaison de quatre caractéristiques (cf. Table 8.2) : lieu d'articulation, degré d'aperture, forme des lèvres et nasalité. On notera que le changement de la valeur d'une seule caractéristique permet souvent de passer d'une voyelle à une autre. Les exemples que nous donnons illustrent ce fait dans la mesure du possible :

- Lieu d'articulation : il s'agit du lieu de la bouche où se positionne la langue. Ce lieu peut être l'avant du palais ([i]), l'arrière du palais ([y]) ou le voile du palais ([u]).
- Degré d'aperture : il s'agit de la hauteur de la langue dans la bouche. Pour un même lieu d'articulation, il y a quatre degrés d'aperture, le premier étant le plus haut ([i]) et le quatrième, le plus bas ([a]).
- Forme des lèvres : les lèvres peuvent être étirées ([i]) ou arrondies ([y]).
- Nasalité : selon les voyelles, l'air passe exclusivement par la bouche ([ɛ], [ɔ]) ou passe également *en partie* par le nez lorsque le voile du palais s'abaisse ([ɛ̃], [ɔ̃]).

La consonne. La caractéristique principale qui différencie une consonne d'une voyelle est l'*obstruction* partielle ou totale du conduit vocal lors de sa prononciation. Cette obstruction engendre des turbulences. Une consonne est de ce fait un *bruit*, qui dans certains cas s'accompagne d'un son si les cordes vocales vibrent. Quatre caractéristiques articulatoires définissent une consonne (cf. Table 8.3) : mode d'articulation, lieu d'articulation, voisement et nasalité. Comme dans le cas des voyelles, le changement de la valeur d'une seule caractéristique permet souvent de passer d'une consonne à une autre :

- Mode d'articulation : il définit la manière dont l'air s'échappe de l'appareil phonatoire. Les consonnes se répartissent principalement en *occlusives*, *constrictives* et *liquides*. Une consonne occlusive, comme [p] ou [t], est caractérisée par une fermeture complète de l'appareil phonatoire en un point, où l'air est accumulé avant d'être brusquement relâché. Une consonne constrictive, comme [s] ou [ʃ], est caractérisée par une forte réduction de l'appareil phonatoire en un point, où l'air s'échappe en étant simplement comprimé. Une consonne liquide, comme [l] ou [ʁ], est caractérisée par un rapprochement modéré des organes phonatoires, de sorte que le bruit caractéristique de friction des constrictives n'est pas produit.
- Lieu d'articulation : il s'agit du lieu de l'appareil phonatoire où le mode d'articulation est réalisé. Ce lieu d'articulation peut varier des lèvres (bilabial, [b]) à la luette (uvulaire [5], [ʁ]).
- Voisement : il dépend des cordes vocales, qui vibrent ou ne vibrent pas. Lorsqu'elles ne vibrent pas, on parle de *consonne sourde* ([p], [t]), et lorsqu'elles vibrent, de *consonne sonore* ([b], [z]).
- Nasalité : comme dans le cas des voyelles, l'air dans certains cas passe exclusivement par la bouche ([b], [d]) ou également en partie par le nez ([m], [n]). En français, toutes les consonnes nasales sont voisées.

La semi-voyelle. Une semi-voyelle possède deux lieux d'articulation, entre lesquels les organes phonatoires se déplacent en cours de prononciation. L'un des lieux est celui d'une voyelle, et l'autre est celui d'une consonne. La semi-voyelle [w], par exemple, est le résultat de la double articulation *vélaire* (comme [u]) et *bilabiale* (comme [b]). Hormis ce double lieu d'articulation, une semi-voyelle se définit selon les mêmes caractéristiques qu'une voyelle (cf. Table 8.4) : degré d'aperture, forme des lèvres et nasalité.

Notons que la littérature parle parfois de *semi-consonne*. Cependant, l'absence d'obstruction du conduit vocal nous pousse à préférer l'appellation de semi-voyelle.

8.3.1.3 Le phonème dans le discours

Dans le discours, un phonème donné possède plusieurs variantes, appelées *allophones*. Un allophone est une *réalisation acoustique* d'un phonème, et possède certains traits

[5] En latin, la luette se dit *uvula*.

Mode	**Vois.**	**Lieu**											
		bilabial		Labiodental	D ental		Alvéolaire	Prépalatal	Palatal	Vélaire		Uvulaire
occlusif	sourd	p			t					k		
	sonore	b	m		d	n			ɲ	g	ŋ	
constrictif	sourd			f			s	ʃ				
	sonore			v			z	ʒ				
Liquide	sonore						l					ʁ
Nasalité		oral	nasal	oral	oral	nasal	oral	oral	nasal	oral	nasal	oral

Tab. 8.3: Description articulatoire des consonnes du français

Aperture	**Lieu**		
	Palatal + Palatal antérieur	Bilabial + Palatal postérieur	Bilabial + Vélaire
1	j	ɥ	w
Nasalité	oral		
Lèvres	étirées	arrondies	

Tab. 8.4: Description articulatoire des semi-voyelles du français

qui, contrairement à ceux du phonème, ne sont pas *distinctifs*, mais simplement *combinatoires* ou *individuels*.

Traits combinatoires. Les traits combinatoires d'un allophone sont dus au contexte phonétique dans lequel il se réalise. Par exemple, comme le montre la Figure 8.5, le phonème [a] est actualisé par deux allophones /a/ différents dans *malle* et dans *sac*, parce que les phonèmes qui l'entourent sont différents. Les différences entre ces deux /a/ attribuables au contexte se situent aux frontières des phonèmes, et sont dues au déplacement des organes phonatoires. Ce déplacement n'est pas un processus instantané, mais continu. Entre deux phonèmes, il y a donc une phase de *coarticulation* pendant laquelle le filtre phonatoire passe progressivement d'un phonème à l'autre.

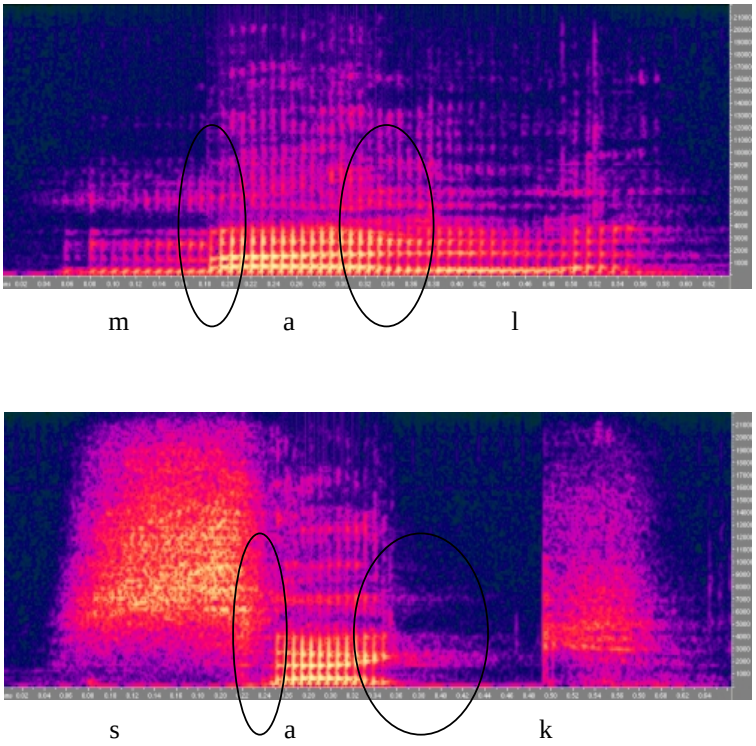


FIG. 8.5: Traits combinatoires dus à la coarticulation. Les zones de coarticulation sont entourées

Du fait de l'importance des phases de coarticulation dans le flux de parole, les spécialistes estiment que la compréhension de la parole repose plus sur les transitions phonétiques que sur les parties stables des phonèmes (Liberman *et al.* 1959). Ce point est d'une importance capitale pour la synthèse de la parole, parce qu'il est à l'origine des difficultés du processus en terme de traitement du signal. Nous y revenons dans la suite de cette partie.

Traits individuels. Les traits individuels sont le fait du locuteur, qui s'écarte plus ou moins de la *norme* en fonction de ses propres compétences articulatoires et du milieu socio-culturel auquel il appartient.

L'assimilation est l'un de ces traits. Il s'agit d'une modification phonétique d'un son due au contexte dans lequel il se situe. L'assimilation est dite *régressive* si un son est influencé par son contexte droit, et *progressive* si un son est influencé par son contexte gauche. Les assimilations les plus fréquentes sont certainement l'assourdissement, la sonorisation et la nasalisation :

- *Assourdissement.* L'assourdissement est le passage d'un son voisé au son non voisé correspondant, généralement en contexte non voisé. Par exemple, « a<u>b</u>solu » [a b s ɔ l y] prononcé /a p s ɔ l y/.
- *Sonorisation.* La sonorisation est le passage d'un son non voisé au son voisé correspondant, généralement en contexte voisé. Par exemple, « anecdote » [a n ɛ k d ɔ t] prononcé /a n ɛ g d ɔ t/.
- *Nasalisation.* Lors de la prononciation d'une voyelle orale suivie par une consonne nasale, la nasalisation consiste à abaisser trop tôt le voile du palais, *pendant* la prononciation de la voyelle. La voyelle orale devient donc nasale. Par exemple, « m<u>ê</u>me » [m ɛ m] prononcé /m ɛ̃ m/.

D'autres traits individuels dépendent indirectement du contexte, en ceci qu'ils sont dus à la position du son par rapport à l'accent. Les phénomènes les plus fréquents sont ici l'amuïssement et la diphtongaison :

- *Amuïssement.* L'amuïssement consiste en l'atténuation ou la suppression complète d'un phonème. Ce phénomène est particulièrement fréquent en position implosive. Par exemple, « sucre » [s y k ʁ] prononcé /s y k/.
- *Diphtongaison.* La diphtongaison est le changement du timbre d'une voyelle en cours de prononciation. Par exemple, « chant<u>er</u> » [ʃ ɑ̃ t e] prononcé /ʃ ɑ̃ t e j/. Ce phénomène est plus fréquent en position accentuée et lorsque la voyelle a une certaine durée.

Notons que ces traits individuels ne sont souvent que l'*amplification* de traits combinatoires. Les traits individuels sont de ce fait à l'origine de l'évolution phonétique des langues qui, au cours du temps, font entrer dans la norme des transformations phonétiques qui ont une tendance à se généraliser au sein de ses locuteurs. Pour ne citer qu'un exemple, la forme latine *bonus* est devenue en français *bon* parce qu'elle a subi un amuïssement de la finale *-us* et une nasalisation du *o* par le *n* : /b ɔ n u s/ → /b ɔ n/ → /b ɔ̃/.

Et en synthèse ? Le processus de synthèse ne tient pas compte des variantes individuelles : le module de traitement automatique de la langue détermine les phonèmes qui correspondent au texte, et le module de traitement du signal tâche de modéliser les allophones *combinatoires* appropriés.

Notons que la synthèse de la parole a défini les unités acoustiques suivantes, dont l'intérêt sera détaillé dans la suite de cette partie : le phone, le diphone et le demi-phone.

1. Le *phone* fait référence à toute réalisation acoustique d'un phonème, et non à un allophone particulier.
2. le *demi-phone* (cf. Figure 8.6) est une unité acoustique prélevée sur un phone, qui s'étend soit du début au milieu de la partie stable du phone (demi-phone gauche), soit du milieu de la partie stable à la fin du phone (demi-phone droit).
3. Le *diphone* (cf. Figure 8.7) est une unité acoustique commençant au milieu d'un phone et se terminant au milieu du phone suivant. Le diphone s'étend donc d'une partie stable du signal à une autre, et inclut la phase instable de coarticulation entre ces deux parties stables.

Convention 8.3.1. Nous adoptons les conventions suivantes :

- [a] : un phonème,
- /a/ : un phone,
- /a| : un demi-phone droit,
- |a/ : un demi-phone gauche,
- |ab| : un diphone.

8.3.2 La prosodie

La prosodie est une branche de la phonétique qui étudie les *traits phoniques supra-segmentaux*, c'est-à-dire les traits qui ne peuvent être perçus sans les phonèmes et n'existent pas sans eux. Il s'agit principalement de la *quantité*, de l'*accentuation*, du *ton* et de la *tension*. Ces traits dépendent de nombreux facteurs, dont la structure syllabique que nous commençons par décrire.

La syllabe. Une syllabe est un groupement de phonèmes qui se prononcent en une seule émission. Toute syllabe comporte au moins un noyau, ou *nucleus*. En français, le noyau est toujours une voyelle alors que certaines langues, comme l'anglais, possèdent également des consonnes dites *vocalisées* qui jouent le rôle de noyau.

Le noyau peut être précédé par une consonne ou un groupe consonantique et éventuellement une semi-voyelle. L'ensemble constitue l'attaque de la syllabe, ou *onset*, marquée par un positionnement des organes phonatoires. On est en phase explosive, l'énergie augmente.

Le noyau est éventuellement suivi d'une semi-voyelle et d'une ou de plusieurs consonnes. L'ensemble constitue la queue de la syllabe, ou *coda*, marquée par un relâchement des organes phonatoires. On est en phase implosive, l'énergie diminue.

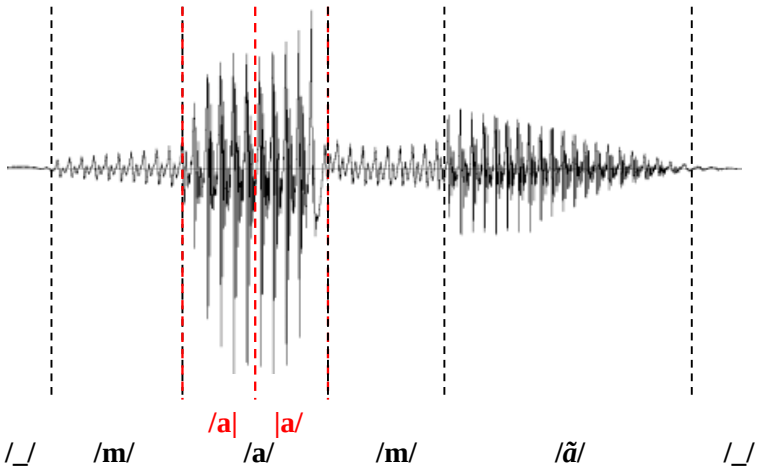


FIG. 8.6: Exemple de demi-phone

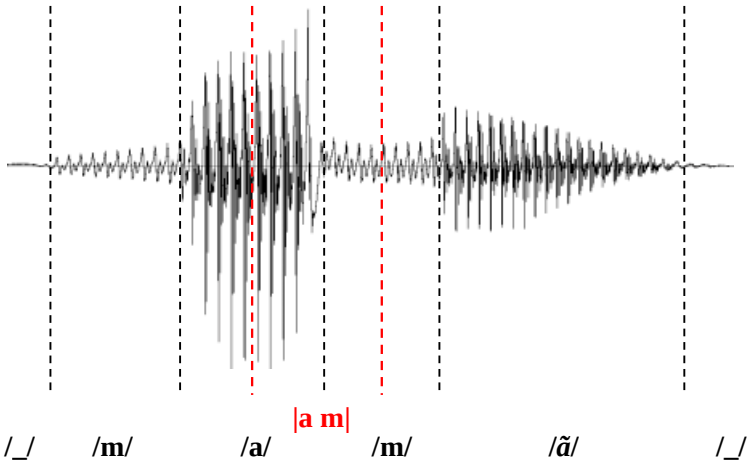


FIG. 8.7: Exemple de diphone

Il existe donc des syllabes de structures et de longueurs différentes. Les syllabes les plus fréquentes en français sont « voyelle » (V), « consonne–voyelle » (CV), « voyelle–consonne » (VC) et « consonne–voyelle–consonne » (CVC).

La quantité. C'est la longueur d'un phonème, décrite de manière discrète. Par exemple, *brève*, *moyenne* et *longue*. La quantité vocalique dépend de plusieurs paramètres, dont la structure syllabique et l'accentuation.

L'accentuation. Il s'agit de la mise en relief d'une syllabe. On distingue généralement l'accent *tonique*, qui consiste en une variation de l'intensité, et l'accent de *hauteur*, qui est

une modification de la hauteur [6] de la voix, bien que souvent l'accent tonique implique une variation de la hauteur également.

Les langues romanes italiques et germaniques occidentales (cf. Introduction, Section 3.2, page 7) possèdent l'accent tonique, qui distingue de ce fait les syllabes toniques des syllabes *atones*, sans accent. L'accent tonique est une caractéristique de la syllabe. Deux accents toniques existent : un accent fort, dit *primaire*, et un accent de moindre importance, dit *secondaire*. L'accent secondaire est posé sur la première syllabe du mot, tandis que la position de l'accent primaire dépend de la langue. En français par exemple, l'accent primaire se trouve toujours sur la syllabe finale du mot, alors qu'en italien et en espagnol, il se place généralement sur la syllabe pénultième, mais peut se déplacer sur l'antépénultième ou sur la finale. Dans certaines langues comme l'italien, l'espagnol ou l'anglais, l'accent de mot est perceptible, ce qui donne à ces langues leur côté chantant. En français par contre, les phonéticiens s'accordent pour considérer que l'accent de mot *disparaît* au profit de l'accent de groupe de mots.

L'accentuation décrite ci-dessus est purement mécanique : elle correspond à une habitude rythmique propre à la langue. Cependant, cette accentuation peut être modifiée afin de mettre en évidence une idée ou une émotion. Une idée est mise en valeur à l'aide d'un accent *intellectif*, qui fait varier la hauteur d'une syllabe. Une émotion sera transmise par un accent *affectif*, principalement réalisé par une modification de l'intensité d'une syllabe. En synthèse classique, les accents intellectif et affectif ne sont pas pris en compte. Ces considérations seront à charge de la synthèse émotionnelle [7], dont les premiers balbutiements se font actuellement entendre (Vine & Sahandi 2000, Black 2003, Schröder 2003).

Le ton. Il s'agit d'une variation de la hauteur de la voix lors de la production d'un phonème. Le ton ressemble donc à un accent de hauteur, si ce n'est que ses variations sont *significatives* : le même mot, selon le ton utilisé, change de sens.

Une langue pourvue du ton est dite *tonale*. Une langue tonale peut également utiliser l'accent tonique, mais celui-ci est secondaire et non significatif. C'est le cas en chinois mandarin, par exemple, langue qui possède quatre tons et un accent tonique. Dans cette langue, le mot « ma » peut signifier *maman*, *chanvre*, *cheval* ou *injurier* selon le ton utilisé.

La tension. La tension dans les muscles articulatoires et les lèvres varie lors de la production des phonèmes. De manière générale, la tension varie fortement d'une langue à l'autre. L'anglais est nettement moins tendu que le français, par exemple. Les voyelles de l'anglais ont de ce fait une tendance naturelle à la diphtongaison.

[6] La hauteur fait référence à la fréquence fondamentale.

[7] Notons que le terme de *synthèse émotionnelle* fait référence à tort aux émotions. Or, une émotion est transmise dans la parole de manière peu contrôlée par le locuteur. La synthèse émotionnelle s'intéressera en réalité à la synthèse d'*attitudes* qui, elles, sont contrôlées par le locuteur, et donc transposables dans un processus de synthèse : attitude positive ou négative, doute, gêne, etc.

La tension varie principalement entre syllabes accentuées et syllabes non accentuées. En anglais par exemple, les voyelles de syllabes non accentuées sont si peu tendues qu'elles ont tendance à être toutes prononcées comme un *schwa* ([ə]), ce *e* dit *moyen central* parce que prononcé au centre de la cavité buccale, à un degré d'aperture moyen et sans arrondissement ni étirement des lèvres.

Et en synthèse ? Les descriptions acoustiques de la Section précédente ont mis en évidence le caractère éminemment mécanique de la parole humaine. Les productions sonores sont soumises aux limites physiques du conduit vocal : rythme de la respiration, tension des cordes vocales, inertie et tension des muscles phonatoires.

Les traits phoniques suprasegmentaux sont au fond une simple *conséquence* de la mise en œuvre de cette mécanique : la respiration impose le placement d'accents et peut allonger certains sons, les muscles peuvent plus ou moins se tendre et l'enchaînement des sons peut avoir une influence sur leur longueur. Selon la langue, les locuteurs ont accentué certaines de ces contraintes mécaniques pour les rendre *significatives*. C'est ainsi que certaines langues favorisent le ton, et d'autres, l'accent tonique. De mécaniques, ces traits suprasegmentaux sont donc devenus porteurs de sens.

Au niveau acoustique, les traits suprasegmentaux correspondent à des traits acoustiques : la quantité détermine une durée, l'accentuation (ton, accent tonique) correspond à une variation de l'énergie et de la fréquence fondamentale du signal, la tension détermine le timbre [8] d'une voyelle. Afin de véhiculer le sens désiré, la synthèse de la parole se doit dès lors de tenir compte de ces traits suprasegmentaux et de les convertir le plus fidèlement possible en valeurs acoustiques.

On comprend maintenant pourquoi le traitement du signal repose sur une représentation phonético-prosodique du texte. Cette représentation est par nature non ambiguë et comporte de nombreuses informations utiles à la génération de signaux acoustiques analogues à ceux produits par l'appareil phonatoire de l'être humain. Le tout, dès lors, est de parvenir à *convertir* cette représentation symbolique en un signal acoustique convaincant pour l'oreille humaine...

8.4 Evolution du concept de synthèse

L'objectif de cette section est d'expliquer comment la notion d'unités non uniformes s'est imposée en synthèse. Elle ne fait donc que survoler les différentes étapes de la synthèse, mettant en évidence, pour chaque étape, les limites qui ont amené à la définition de l'étape suivante. Le lecteur qui serait intéressé par le détail de ces techniques de synthèse se reportera entre autres utilement à (Liberman *et al.* 1959, Klatt 1980, Charpentier & Stella 1986,

[8] Les formants.

Moulines & Charpentier 1990, Pols 1990, Stevens 1990, Dutoit 1993, Moulines & Laroche 1995, Daelemans & van den Bosch 1997, Malfrère *et al.* 1990, Boîte *et al.* 2000).

8.4.1 Vous avez dit *articulatoire* ?

Nous l'avons vu, il est acquis depuis longtemps que la compréhension de la parole repose plus sur les transitions phonétiques que sur les parties stables des phonèmes. Les premiers systèmes de synthèse (Klatt 1980, Pols 1990, Stevens 1990) ont donc tout naturellement tâché de modéliser ces phénomènes de coarticulation au travers de *jeux de règles* décrivant l'influence que les phonèmes ont les uns sur les autres.

Corpus et paramètres. Le processus nécessite la constitution d'un corpus de parole contenant un grand nombre de mots lus par un locuteur professionnel. Les mots, choisis sur la base de leur structure syllabique, sont censés représenter au mieux les phénomènes de coarticulation se produisant aux frontières des phonèmes.

L'analyse du corpus sépare dans un premier temps les contributions respectives de la source glottique [9] et du conduit vocal. Ceci permet dans un second temps de représenter la contribution du conduit vocal sous la forme de paramètres choisis en fonction d'un modèle paramétrique donné, dont l'intérêt est qu'il facilite l'analyse et donc l'établissement des règles. A des fins de synthèse, les paramètres retenus sont au plus les quatre premiers formants et la durée des phases de coarticulation.

Les règles sont construites à partir des paramètres prélevés sur le corpus, à l'aide d'un processus itératif d'essais-erreurs dont l'objectif est d'optimiser les règles afin d'améliorer la qualité de la synthèse.

Synthèse. Dans un tel système, le processus de synthèse consiste à générer les valeurs des paramètres du modèle en fonction de la représentation phonético-prosodique, et de combiner les valeurs obtenues à l'excitation adéquate. Dans le cas de la synthèse d'un son voisé, l'excitation est une onde périodique proche de la forme de l'onde glottique. Dans le cas de la synthèse d'un son non voisé, l'excitation est un bruit blanc, généré de manière pseudo-aléatoire [10] (cf. Figure 8.8).

Limites. La qualité de la synthèse obtenue, en se limitant au système de règles, dépend :

- de la qualité du corpus, c'est-à-dire du choix des mots et de la qualité de l'enregistrement.
- de l'adéquation du modèle paramétrique au signal qu'il modélise. Si les durées coarticulatoires et les premiers formants sont en effet pertinents, ces paramètres ne représentent pas pour autant *toute* la richesse d'un signal de parole. On parle d'*erreur de*

[9] Les cordes vocales.

[10] Notons que ce bruit blanc est en fait périodique, mais son caractère périodique est imperceptible pour autant que sa période soit supérieure à 5 secondes.

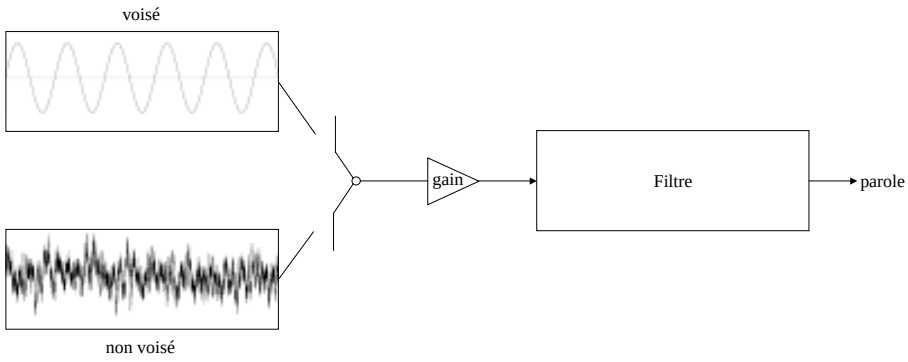


FIG. 8.8: Synthèse par règles

modélisation intrinsèque. Il faut cependant noter que les meilleurs systèmes emploient plus de 60 paramètres, ce qui est déjà beaucoup pour un processus qui se doit de rester temps réel. La modélisation d'autres caractéristiques du signal augmenterait encore le nombre des paramètres nécessaires...

- de l'algorithme d'analyse qui fixe les valeurs des paramètres choisis. On parle d'*erreur de modélisation extrinsèque*.
- de la pertinence des règles déduites de l'analyse des paramètres, qui dépend entre autres de la qualité du processus itératif d'essais-erreurs.

Conclusion. Les systèmes de synthèse par règles, gérant au travers de modèles mathématiques les phénomènes coarticulatoires, permettent théoriquement de générer de la parole de haute qualité (Dutoit 1993). Malheureusement, la parole produite par ce type de systèmes est marquée de bourdonnements gênants, artefacts de synthèse dus aux limites des modèles utilisés et des règles déduites. Les *bons* paramètres et les *bonnes* règles, faut-il supposer, n'ont simplement pas encore été découverts...

8.4.2 Encapsulation dans des unités...

Forts de ce constat, certains chercheurs ont décidé d'encapsuler la phase de coarticulation entre phonèmes, si difficile à modéliser à l'aide de règles, dans des *unités acoustiques*. C'est ainsi qu'est né le concept de *diphone* (cf. Figure 8.7).

Certains phonèmes cependant ne présentent pas de phase stable, comme les semi-voyelles. Dans ce cas, l'encapsulation peut contenir un phonème entier (*triphone*), voire deux (*tétraphone*). On rassemble l'ensemble de ces unités acoustiques sous le terme générique de *polyphones*.

Corpus neutralisé. Un système de synthèse qui utilise ce type d'unités génère le signal de parole en *concaténant* bout à bout les polyphones correspondant aux phonèmes déterminés

par le traitement automatique de la langue. Par exemple, pour synthétiser le mot « maman » ([_ m a m ɑ̃ _] [11]), la synthèse doit concaténer les diphones |_ m|, |m a|, |a m|, |m ɑ̃| et |ɑ̃ _|. En synthèse par concaténation, le système a donc une idée très limitée des données qu'il manipule.

Les premiers systèmes de ce type n'ont conservé qu'un exemplaire de chaque polyphone. Les uns (Moulines & Charpentier 1990) ont conservé les unités telles quelles, tandis que d'autres (Dutoit 1993) ont « neutralisé » les polyphones en leur attribuant une F0 constante. Dans les deux cas cependant, le corpus doit être prononcé par le locuteur de la manière la plus neutre *possible*. Pour faciliter la neutralité du ton du locuteur,

- chaque polyphone est placé au centre d'un mot, de sorte que les accents initiaux et finaux, inhérents à la mécanique de la parole, n'affectent pas l'unité.
- chaque mot est un *logatome*. En synthèse, un logatome est un mot *sans signification* pour le locuteur qui le prononce. L'intérêt du logatome est qu'il permet au locuteur d'*essayer* de le prononcer de manière neutre, sans ajouter de marques prosodiques liées à ce que le terme lui évoque.

Synthèse. Comme le montre le schéma de la Figure 8.9, dans un tel système (Charpentier & Stella 1986, Moulines & Charpentier 1990, Dutoit 1993), le module de traitement du signal prend directement les polyphones désirés dans la base, sans phase de sélection. Le module doit ensuite :

- Appliquer à chaque polyphone la courbe prosodique désirée. Ceci implique un traitement du signal sur la totalité de l'unité, puisqu'il est question de modifier la F0 dans le cas d'un demi-phone voisé, et qu'il est nécessaire d'étirer ou de raccourcir chaque demi-phone en fonction de la durée totale à appliquer au phonème correspondant.
- Concaténer les unités, ce qui nécessite d'assurer un *raccord de qualité* aux frontières des unités. Le principe, généralement, est de chercher un point du signal où les unités ne présentent pas de déphasage [12]. A ce point, on effectue un lissage spectral des signaux, de sorte que le passage d'une unité à l'autre s'effectue sans discontinuité audible. Pour deux diphones D_1 et D_2 à concaténer, le lissage consiste à calculer, sur un intervalle court, une moyenne pondérée des valeurs des deux signaux. Puisqu'il s'agit d'un intervalle de durée finie, la pondération est réalisée à l'aide d'une fonction fenêtre, comme la fenêtre de Hanning.

La Figure 8.10 donne un exemple de résultat obtenu avec le système présenté dans (Dutoit 1993).

[11] [_] représente le silence et correspond aux frontières de la phonation ou à une pause en cours de phonation.

[12] La phase est la valeur instantanée d'une grandeur qui évolue de manière cyclique. Le déphasage entre deux signaux est la différence entre leurs phases, qui peut se mesurer comme un angle sur le cercle trigonométrique.

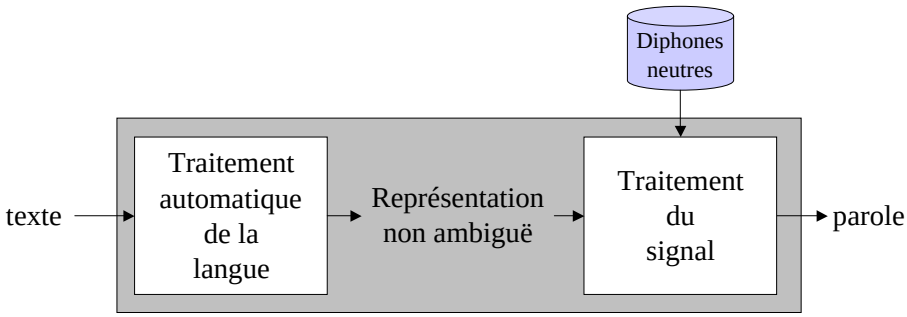


FIG. 8.9: Synthèse par concaténation

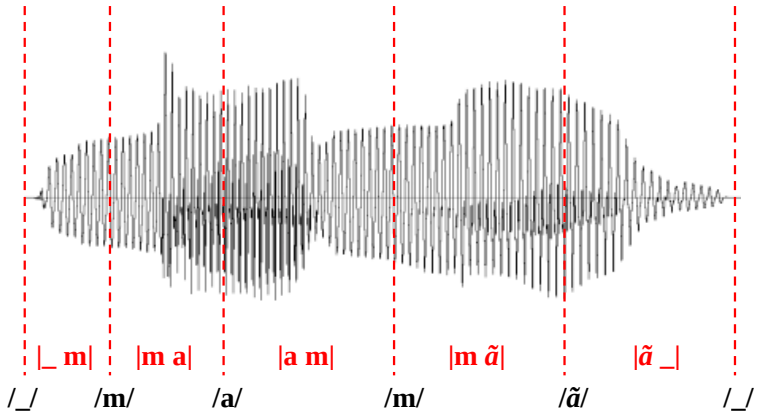


FIG. 8.10: Synthèse par concaténation du mot « maman »

Limites. La qualité de la synthèse obtenue, en se limitant aux effets dus au module de traitement du signal, dépend :

- de la qualité du corpus, c'est-à-dire du choix des logatomes, de la qualité de l'enregistrement et du degré de neutralité avec lequel le locuteur a prononcé chaque logatome.
- de l'adéquation du modèle paramétrique au signal qu'il modélise. Il faut donc que les paramètres extraits représentent toute la richesse du spectre vocal.
- de l'algorithme de resynthèse qui applique F0 et durée aux unités.
- de l'algorithme de lissage des frontières.

Conclusion. Le résultat de la synthèse par concaténation d'unités neutralisées est une parole de synthèse de bonne qualité, dont les problèmes de coarticulation ont disparu, mais qui conserve un côté synthétique qui varie, selon les synthétiseurs, du nasillard au métallique. Ce côté synthétique est évidemment dû au traitement du signal conséquent, mais nécessaire à l'application des valeurs prosodiques choisies. La parole générée, qui est tout à fait com-

préhensible et aide entre autres les personnes handicapées à communiquer et à accéder à l'information, est de ce fait encore peu naturelle.

Discussion. Turing (1950) a proposé ce que l'on appelle maintenant le *test de Turing*. Ce test consiste à confronter un être humain à deux interlocuteurs qu'il ne voit pas. L'un des interlocuteurs est un humain et l'autre, un ordinateur. Si à la fin du test, l'être humain ne peut dire lequel de ses interlocuteurs est la machine, on considère que la machine a passé le test avec succès : elle *raisonne* comme un être humain.

A l'époque de Turing, la synthèse de la parole n'existait pas encore. Les interactions devaient donc se dérouler *par écrit*. De nos jours, ce test pourrait se dérouler de manière orale, et l'on pourrait imaginer que son objectif ne soit pas d'estimer les capacités de raisonnement de la machine, mais ses capacités d'élocution. Si un tel test était réalisé, les synthétiseurs par concaténation d'unités neutralisées ne permettraient pas encore à la machine de le passer avec succès.

Ceci a poussé les chercheurs à se tourner vers une autre voie, privilégiant le corpus au traitement du signal... La voie de la sélection d'unités non uniformes.

8.5 Les principes de la synthèse par sélection

Le problème majeur de la synthèse par concaténation est évidemment la neutralisation des unités, puisque cela implique un traitement du signal conséquent qui diminue le naturel de la parole.

L'idée a dès lors été de constituer un corpus d'unités acoustiques non neutralisées, contenant non plus un seul, mais plusieurs exemplaires de chaque unité. L'intérêt d'un tel corpus, *a priori*, est qu'il permet de mettre en œuvre le principe de *choose the best to modify the least* (Balestri *et al.* 1999) : le but est que la *meilleure suite* d'unités possible soit *sélectionnée*, afin de réduire le traitement du signal au strict minimum, en espérant que ceci confère à la parole de synthèse un caractère plus naturel. Comme l'illustre la Figure 8.11, un système de ce type nécessite dès lors une étape de *sélection* entre les étapes de traitement de la langue et de traitement du signal : à l'aide de certains critères dont les valeurs ont été déterminées par le traitement de la langue, la sélection choisit les unités de parole les plus appropriées, que le traitement du signal devra ensuite prélever du corpus et concaténer pour produire le signal de parole.

La qualité d'un tel système dépend dès lors de deux facteurs : la qualité du corpus de parole et la manière dont la sélection est menée.

8.5.1 Le corpus de parole

La construction d'un corpus de parole pour la synthèse par sélection est un véritable sujet d'étude en soi (Buchsbaum & van Santen 1996, van Santen & Buchsbaum 1997, Black & Lenzo 2001, Kominek & Black 2004) et implique de tenir compte des éléments suivants :

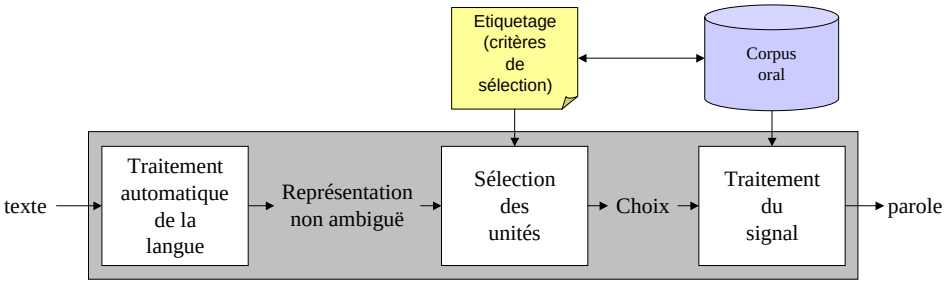


FIG. 8.11: Synthèse par sélection d'unités non uniformes

1. L'unité de base.
2. La méthode de construction de la version textuelle du corpus.
3. Le lieu d'enregistrement du corpus oral.
4. Le choix du locuteur.
5. La direction du locuteur au cours de l'enregistrement.
6. L'alignement et l'étiquetage du corpus oral.

8.5.1.1 L'unité de base

Dans l'optique de choisir la meilleure suite d'unités acoustiques, il est nécessaire de construire un corpus qui contienne une bonne couverture des *variations acoustiques* de la langue. Pour assurer cette couverture, il faut :

1. Choisir une liste de critères susceptibles de couvrir les variations acoustiques. Le nombre de combinaisons possibles des critères retenus correspond au produit de leur nombre respectif de valeurs :

$$K = \prod_{i=1}^{n} |F_i| \tag{8.5.1.1}$$

où n est le nombre de critères, F_i est le i^e critère et $|F_i|$ note le nombre de valeurs de ce critère.

2. Choisir une unité de base. Trois candidats viennent naturellement à l'esprit : le phone, le demi-phone et le diphone. Pour une langue contenant m phonèmes, le nombre d'exemplaires uniques de chaque unité vaut :

   - Phone : m
   - Demi-phone : $2m$
   - Diphone : m^2

Prenons un exemple. Si l'on retient 5 caractéristiques comptant respectivement 2, 4, 3, 2 et 5 valeurs pour une langue de 36 phonèmes, on aura besoin d'un corpus de la taille suivante :

$$\left.\begin{array}{ll} \bullet \text{ Phone :} & 36 \\ \bullet \text{ Demi-phone :} & 2*36 \\ \bullet \text{ Diphone :} & 36^2 \end{array}\right\} * 2 * 4 * 3 * 2 * 5 = \left\{\begin{array}{r} 8\,640 \\ 17\,280 \\ 311\,040 \end{array}\right. \text{ unités}$$

On constate que la taille d'un corpus de diphones devient vite prohibitive, raison pour laquelle les chercheurs (van Santen & Buchsbaum 1997, Black & Lenzo 2001) ont préféré le juger inadapté à la synthèse par sélection. Les systèmes emploient dès lors le phone ou le demi-phone.

8.5.1.2 Analyse et construction du corpus textuel

Le corpus de sélection est un corpus *textuel lu*. Ainsi, le corpus de sélection est *d'abord* un corpus écrit avant d'être un corpus oral. Il est donc nécessaire d'*estimer* les variations acoustiques présentes dans le corpus de sélection sur sa version textuelle, avant de l'enregistrer.

Pour ce faire, et afin d'assurer une compatibilité et une cohérence entre le corpus de sélection et le système de sélection, l'estimation des variations acoustiques doit être réalisée *en fonction des critères retenus pour la sélection* : des critères de type linguistique ou acoustique. Les outils d'analyse, en toute logique, doivent être ceux présents dans le système de synthèse.

A terme, l'objectif est bien sûr de posséder le corpus textuel susceptible de présenter l'ensemble des variations acoustiques *nécessaires*. Il n'est cependant pas question de prendre n'importe quel texte. Il faut le corpus *le plus compact possible*, au vu de la complexité de l'alignement du corpus oral (cf. Section suivante) et de la place disque nécessaire au stockage de la parole. Le principe général est dès lors de *construire* le corpus de sélection à partir d'un vaste ensemble de textes, le corpus global. Le corpus de sélection est ainsi la partie la plus pertinente, à la fois nécessaire et suffisante, du corpus global.

L'algorithme couramment mis en place est qualifié de *greedy algorithm* dans la littérature anglaise. Comme l'indique son nom, cet algorithme est « gourmand ». Il nécessite beaucoup de ressources et de temps : on avoisine le Go de mémoire vive utilisée, et le processus s'étend sur plusieurs heures, voire plusieurs jours selon la quantité de données à analyser. Son principe, par contre, est fort simple (van Santen & Buchsbaum 1997, Black & Lenzo 2001). L'idée est d'utiliser un système de pondération des phrases du corpus global, de manière à pouvoir les classer. L'algorithme est par nature récursif :

1. Analyse de toutes les phrases du corpus global.
2. Pondération des phrases du corpus global.
3. Ajout de la meilleure phrase du corpus global au corpus de sélection et suppression de cette phrase du corpus global.
4. Repondération des phrases *encore présentes* dans le corpus global.

5. Si le corpus global ne contient plus de phrase utile, le processus s'arrête. Sinon, retour au point 3.

Les critères de pondération principaux sont les unités de la sélection. Une unité est donc une unité acoustique de base (demi-phone, phone) accompagnée des critères (linguistiques ou acoustiques) retenus pour la sélection. Pour une phrase donnée, chacune des unités présentes est identifiée. On compare ensuite les unités identifiées aux unités *utiles* au système. La pondération d'une phrase est donc principalement un rapport entre le nombre total d'unités et le nombre d'unités utiles qu'elle contient. Cette pondération peut évidemment intégrer d'autres critères, comme la longueur de la phrase. En général, on favorisera les phrases de longueur moyenne.

A chaque itération, la sélection de la meilleure phrase du corpus implique de mettre à jour une liste des unités encore utiles : de cette liste, toutes les unités de la phrase sélectionnée sont supprimées. Ceci explique que l'algorithme doive repondérer l'ensemble des phrases non sélectionnées à chaque itération, puisque le nombre d'unités utiles de chaque phrase peut avoir changé.

Nous l'avons noté, la condition d'arrêt est que le corpus global ne contienne plus de phrase utile. Ceci signifie que toutes les unités présentes dans les phrases du corpus global ont déjà été ajoutées au corpus de sélection. Ceci, par contre, ne signifie pas que *toutes les unités utiles* aient été trouvées. Selon le nombre des critères retenus et la qualité du corpus global utilisé, il se peut en effet que toutes les unités ne puissent être trouvées. Dans ce cas, la méthode consiste à créer de toute pièce des phrases qui contiennent les unités absentes.

8.5.1.3 Lieu d'enregistrement

Afin d'éviter tout bruit parasite dans l'enregistrement, l'idéal est évidemment de réaliser celui-ci dans une chambre anéchoïque [13], comme un studio d'enregistrement. Cependant, Prudon & d'Alessandro (2001) ont réalisé des enregistrements de qualité acceptable dans une pièce calme.

8.5.1.4 Choix du locuteur

Le corpus ne doit pas être neutralisé, mais il ne peut contenir n'importe quelle variation prosodique. En effet, le but premier du corpus est d'assurer une continuité prosodique et une fluidité des unités concaténées dans la parole de synthèse, tout en respectant la *norme* de la langue. Ceci signifie que la parole naturelle du corpus doit être prononcée :

- en respectant les normes de la langue (timbre, durée et tension des voyelles, etc.),
- sur un ton et avec un débit les plus constants possible,

[13] Une chambre anéchoïque est une pièce dont les parois absorbent totalement les ondes sonores ou électromagnétiques. Une telle pièce permet d'éviter la production de réverbérations qui perturbent la prise de mesures acoustiques et l'enregistrement sonore.

- sans adjonctions prosodiques dues au sens de l'énoncé.

Il est donc nécessaire de recourir à un locuteur professionnel, capable de contrôler les paramètres de son élocution.

8.5.1.5 Direction du locuteur

Il ne suffit pas que le locuteur soit un professionnel compétent et que le lieu d'enregistrement soit de qualité pour que le résultat de l'enregistrement corresponde au corpus désiré. Encore faut-il que le locuteur réalise effectivement les unités attendues. Or, les unités attendues dans une phrase donnée du corpus textuel ont été déterminées par le traitement automatique de la langue, qui *suppose* que le texte correspond à cette suite d'unités. Le locuteur, quant à lui, ne fera peut-être pas la même analyse, et ce pour différentes raisons :

- le traitement automatique de la langue s'est trompé : ses informations linguistiques ne sont pas complètes, ou l'analyse syntaxique est erronée,
- le locuteur lui-même s'est trompé : par exemple, il ne connaît pas un mot (un nom propre, etc.),
- les deux points de vues sont justifiables, mais différents. Par exemple, le timbre d'une voyelle peut être ouvert pour l'un et fermé pour l'autre, comme une alternance [ɛ] – [e], parce que l'un a un point de vue phonologique et l'autre, phonétique.

Les unités produites peuvent dès lors s'écarter des unités attendues. Il est donc nécessaire de guider le locuteur, à l'aide de méta-informations agrémentant le texte à prononcer.

8.5.1.6 Alignement et étiquetage du corpus oral

Lorsque l'enregistrement a été réalisé, il faut encore déterminer où commencent et terminent les unités de chaque phrase dans le corpus oral. Pour ce faire, on utilise un programme d'alignement, basé sur un système de reconnaissance de la parole. Le principe est de présenter au système de reconnaissance le signal de parole d'un côté et la séquence de phones qui y correspond de l'autre. L'alignement obtenu est un *compromis* entre les probabilités émises par le système de reconnaissance et la nécessité de respecter la séquence de phones demandée. Il n'est donc pas *parfait*. Au mieux, et moyennant un sur-entraînement du système de reconnaissance sur la voix du locuteur enregistré, l'alignement obtenu avoisine les 95% en acceptant un écart de 20 ms entre le bon alignement et l'alignement automatique. Ceci demande quelques commentaires :

- L'alignement est réalisé au niveau du phone. Or, un phone commence et termine par deux phases de coarticulation. Il est donc relativement difficile, même pour l'humain averti (un phonéticien ou un acousticien), de déterminer le lieu précis où se termine un phone et commence le phone suivant. Les alignements proposés par deux aligneurs humains ne sont d'ailleurs identiques, au mieux, qu'à environ 80%. Ceci justifie qu'un écart de 20 ms soit autorisé (Sjölander 2001, Paulo & Oliveira 2004).

- *A priori*, l'alignement obtenu doit néanmoins être corrigé manuellement, ce qui représente un travail de longue haleine : comme nous le détaillons plus loin dans ce Chapitre, un aligneur humain averti prend environ 45 jours pour aligner une seule heure de parole...

Lorsque l'alignement désiré est obtenu, une table d'étiquetage est construite. Elle contient, pour chaque unité, les informations nécessaires à la sélection, dont bien sûr les durées des phones déduites de l'alignement. Comme le montre le schéma de la Figure 8.11, la table d'étiquetage sera principalement utilisée par le modèle de sélection, tandis que le corpus oral sera principalement utilisé par le module de traitement du signal.

8.5.2 Le processus de sélection

L'étape de sélection doit choisir dans le corpus de parole les unités qui correspondent au mieux aux unités décrites par l'analyse de la langue, tout en vérifiant que celles-ci se concatènent sans trop de discontinuités acoustiques. Cette étape comprend donc une phase de pré-sélection avant la phase de sélection proprement dite.

8.5.2.1 Pré-sélection

Nous avons expliqué que les unités du corpus de parole sont étiquetées par une liste de critères choisis, de manière à être identifiables. Dans une phase de pré-sélection pour une cible donnée, le système recherche donc dans le corpus les unités acoustiques selon cette liste de critères. Cependant, comme Breen & Jackson (1998a) le constatent très justement, « *if a system existed which contained a complete inventory of sounds, then an unordered list of adequate features would be sufficient to select the desired sound. Unfortunately this is not the case, so some method of assigning relative importance to individual attributes must be devised.* » La méthode classiquement retenue est une pondération des critères. L'ensemble {critères, pondérations} constitue un *coût cible* qui, pour une cible c_1 et une unité candidate u_1, se calcule comme la somme des distances pondérées entre les valeurs de c_1 et de u_1 pour les critères retenus :

$$\texttt{CI}(u_j|c_j) = \sum_{i=1}^{n} w_i \times D(u_j^i, c_j^i) \tag{8.5.2.1}$$

où n est le nombre de critères retenus pour le coût cible, i est le i^e critère cible, w_i est le poids attribué à ce critère, c_j^i est la valeur de ce critère pour la cible, u_j^i est la valeur de ce critère pour l'unité candidate, et $D(c_j^i, u_j^i)$ note la distance entre ces valeurs. Selon le critère, la distance peut être une valeur binaire (0, 1), discrète (*e.g.*, 0, 0.25, 0.5, 0.75, 1) ou continue.

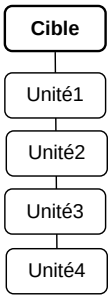


FIG. 8.12: Liste de candidats pour une unité cible

8.5.2.2 Sélection

Les candidats choisis pour l'ensemble des unités cibles d'une phrase forment un treillis de solutions possibles, comme le montre le schéma de la Figure 8.13. L'étape de sélection doit retenir uniquement la meilleure suite d'unités. Cette meilleure suite n'est pas forcément la liste des candidats les plus proches des cibles. La sélection se doit d'éviter les discontinuités audibles entre unités contiguës dans le signal. Ceci implique de calculer un *coût de concaténation* en fonction d'informations, *a priori* acoustiques, prélevées sur le corpus. Ici également, on peut faire l'hypothèse que parmi les critères choisis, certains seront plus discriminants. Une pondération est donc également nécessaire. Le coût de concaténation entre deux unités u_1 et u_2 contiguës se calcule donc comme la somme des distances pondérées entre les valeurs des deux unités pour les critères retenus :

$$\texttt{CO}(u_{j-1}, u_j) = \sum_{i=1}^{m} w_i \times D(u_{j-1}^i, u_j^i) \tag{8.5.2.2}$$

où m est le nombre de critères retenus pour le coût de concaténation, i est le i^e critère de concaténation, w_i est le poids attribué à ce critère, u_j^i est la valeur de ce critère pour l'unité u_j, u_{j-1}^i est la valeur de ce critère pour l'unité u_{j-1}, et $D(u_{j-1}^i, u_j^i)$ note la distance entre ces valeurs.

La synthèse par sélection repose donc sur le calcul d'un double coût, *coût cible* et *coût de concaténation*. A ce niveau aussi, on peut supposer qu'une pondération peut être nécessaire, de manière à pouvoir favoriser la proximité prosodique (coût cible) ou la fluidité acoustique (coût de concaténation). La suite d'unités U qui sera sélectionnée sera celle qui minimisera ce double coût pondéré :

$$U = \arg\min \sum_{j=1}^{l} W_{\texttt{CI}} \times \texttt{CI}(u_j|c_j) + W_{\texttt{CO}} \times \texttt{CO}(u_{j-1}, u_j) \tag{8.5.2.3}$$

où l est le nombre d'unités de la suite, $W_{\texttt{CI}}$ est le poids accordé au coût cible, et $W_{\texttt{CO}}$ est le poids accordé au coût de concaténation.

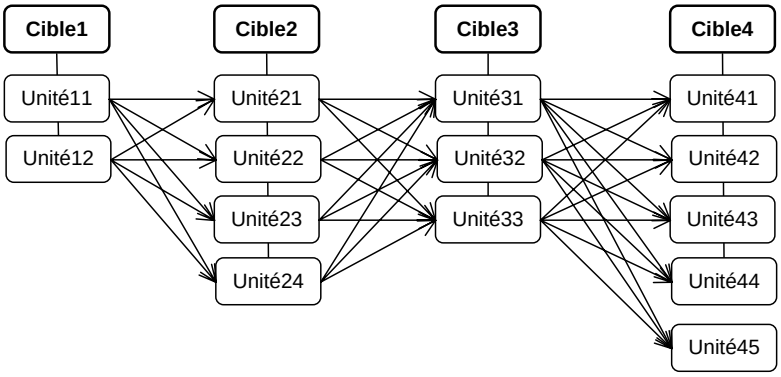


FIG. 8.13: Treillis de solutions

8.5.2.3 Unités non uniformes

Quelle que soit l'unité de base choisie, les unités de ce type de synthèse sont de manière générale qualifiées de *non uniformes* (Sagisaka 1988). Il y a deux raisons à cela. La première, les différentes occurrences d'une même unité diffèrent *au niveau acoustique*, ce qui est nécessaire afin d'obtenir la courbe prosodique désirée en minimisant le traitement du signal. La seconde, les unités sélectionnées peuvent varier *en longueur*. En effet, si la plupart des systèmes pré-sélectionnent à l'aide de l'unité de base retenue, la sélection peut favoriser les unités contiguës dans le corpus, ce qui revient à choisir une syllabe, un mot, un syntagme ou une phrase complète. Notons que certains systèmes cherchent effectivement des unités de longueur différente dès la phase de pré-sélection (cf. Section 9.1).

8.5.2.4 Optimisation

Les étapes de pré-sélection et de sélection impliquent de coûteuses recherches et de nombreux calculs. Or, étant donné que la synthèse est un processus utilisé dans de nombreux systèmes interactifs, il est nécessaire que le processus reste « temps réel ».

Définition 8.5.1 (Temps réel)**.** *Un processus* temps réel *est un processus dont le temps de traitement est adapté à l'évolution de ce qu'il contrôle* (Liu 2000)*. En synthèse de la parole, il est communément admis que le processus est temps réel si le temps de traitement d'un texte à synthétiser est équivalent au temps nécessaire à le prononcer.*

Selon cette définition, un processus temps réel n'est donc pas instantané : un signal de 10 secondes, par exemple, demande 10 secondes de traitement. Rester « au moins » temps réel est cependant une nécessité, qui implique l'utilisation d'une méthode d'optimisation de la recherche des unités.

8.5.3 Analyse

La qualité d'un système de synthèse par sélection dépend fortement de la qualité du corpus et de la finesse de la sélection. La qualité du corpus est déterminée par un ensemble de facteurs *extrinsèques* au système :

1. Le choix des phrases qui le constituent.
2. L'environnement de l'enregistrement.
3. Le choix du locuteur et sa performance au moment de l'enregistrement.
4. L'aide et le support proposés au locuteur.
5. La qualité de la segmentation obtenue.

La finesse de la sélection dépend de facteurs *intrinsèques* au système :

1. Les critères de sélection, c'est-à-dire :
   (a) Les critères du coût cible et leur pondération.
   (b) Les critères du coût de concaténation et leur pondération.
   (c) La pondération globale entre coût cible et coût de concaténation.
   (d) Le degré de traitement du signal appliqué aux unités sélectionnées en fonction des résultats de la sélection.
2. La méthode d'optimisation du processus de sélection.

8.6 Conclusion

Partant du fait que la compréhension de la parole repose plus sur les transitions phonétiques que sur les parties stables des phonèmes, les premiers systèmes de synthèse ont tâché de modéliser les phénomènes de coarticulation au travers de jeux de règles décrivant l'influence que les phonèmes ont les uns sur les autres. Ces systèmes, gérant au travers de modèles mathématiques les phénomènes coarticulatoires, permettent théoriquement de générer de la parole de haute qualité. Malheureusement, la parole produite par ce type de systèmes est marquée de bourdonnements gênants, *artefacts* de synthèse dus aux limites des modèles utilisés.

Forts de ce constat, certains chercheurs ont décidé d'encapsuler la phase de coarticulation entre phonèmes, si difficile à modéliser à l'aide de règles, dans des unités acoustiques. C'est ainsi qu'est né le concept de *diphone*. Le résultat de la synthèse par concaténation d'unités neutralisées est une parole de synthèse de bonne qualité, dont les problèmes de coarticulation ont disparu, mais qui conserve un côté synthétique qui varie, selon les synthétiseurs, du nasillard au métallique. La parole générée, qui est tout à fait compréhensible et aide entre autres les personnes handicapées à communiquer et à accéder à l'information, est cependant encore peu naturelle.

Ceci a poussé les chercheurs à se tourner vers une autre voie, privilégiant le corpus au traitement du signal : la voie de la sélection d'unités non uniformes. L'idée de ce type de

synthèse, basée sur un corpus contenant plusieurs exemplaires d'unités acoustiques non neutralisées, est de sélectionner la meilleure suite d'unités possible, afin de réduire le traitement du signal au strict minimum, en espérant que ceci confère à la parole de synthèse un caractère plus naturel. Un tel système nécessite donc une étape de sélection des unités, qui doit choisir dans le corpus de parole les unités qui correspondent au mieux aux unités décrites par l'analyse de la langue, tout en vérifiant que celles-ci se concatènent sans trop de discontinuités acoustiques. Cette étape comprend donc une phase de pré-sélection avant la phase de sélection proprement dite, et réalise l'évaluation d'un double coût « cible-concaténation ».

L'analyse des principes de la synthèse par sélection d'unités non uniformes met en évidence que les performances d'un tel système dépendent fortement de la qualité du corpus de parole et de la finesse de la méthode de sélection. Ce sont ces points que nous abordons au chapitre suivant, qui présente les systèmes qui font l'état de l'art en synthèse par sélection d'unités non uniformes...

Chapitre 9

Etat de l'art en synthèse par sélection

Les différents systèmes de l'état de l'art sont présentés dans (Sagisaka 1988, Black & Campbell 1995, Hunt & Black 1996, Taylor & Black 1997, Breen & Jackson 1998a,b, Balestri *et al.* 1999, Beutnagel *et al.* 1999a,b, Taylor & Black 1999, Taylor 2000, Yi *et al.* 2000, Bulyko & Ostendorf 2001, Prudon & d'Alessandro 2001, Allauzen *et al.* 2004). Ils possèdent à peu près tous leurs propres voix. En outre, rares sont les détails donnés par les auteurs sur les conditions de l'enregistrement, les critères de choix du locuteur et le guidage réalisé. Il est dès lors difficile de les comparer sur la qualité du corpus.

Notre travail a quant à lui essentiellement consisté à la mise au point d'une nouvelle méthode de sélection et à sa modélisation à l'aide de machines à états finis. Nos tests reposent quant à eux sur un corpus pré-existant et non sur un corpus construit expressément pour la méthode.

De ce fait, notre analyse des systèmes de l'état de l'art se limite à leurs facteurs intrinsèques. Cette analyse se divise en deux points. La Section 9.1 aborde les critères retenus pour la sélection, et la Section 9.2 s'intéresse aux méthodes d'optimisation du processus de sélection.

Des échantillons de synthèse de plusieurs systèmes de l'état de l'art sont disponibles sur notre site [1].

9.1 Critères de sélection

Cette section commence par rappeler l'approche du tout premier système qui, dédié au Japonais, a permis de jeter les bases de la sélection d'unités non uniformes, mais n'était pas transposable aux langues à prosodie riche.

Nous abordons ensuite les systèmes adaptés aux langues à prosodie riche qui ont jalonné l'évolution du concept de sélection d'unités. Ces systèmes se répartissent en deux catégories selon les critères retenus pour la sélection. Les premiers – souvent les plus anciens – ont employé un maximum d'informations acoustiques accompagnées de quelques

[1] `richardbeaufort.co.nr/phd/2-nuu-sota.html`.

informations linguistiques ; nous les qualifions de *systèmes acoustiques*. Les seconds – majoritairement les plus récents – ont limité l'utilisation de valeurs acoustiques autant que faire se peut, en faveur de données purement linguistiques ; nous les qualifions de *systèmes linguistiques*.

9.1.1 Les bases

Le système développé par Sagisaka (1988, 1992), dédié au Japonais, est le premier à proposer l'emploi d'unités non uniformes en synthèse de la parole, au moment où la synthèse par concaténation est seulement en train de prendre forme (Charpentier & Stella 1986, Moulines & Charpentier 1990, Dutoit 1993). Ses caractéristiques principales sont les suivantes :

Le système emploie un corpus d'un peu plus de 5 000 mots représentés sous la forme d'un arbre phonétique qui indexe toutes les séquences et sous-séquences phonétiques de ces mots. Par exemple, si le corpus contient les mots « n*aga*sa » et « un*aga*su », la sous-séquence phonétique « aga » sera indexée et pointera vers ces deux mots.

Lors de la synthèse d'une séquence phonétique (« hana » par exemple), l'interrogation de l'arbre phonétique permet de retrouver les unités de la séquence en elle-même et de toutes ses sous-séquences gauches (« han- », « ha- », « h- ») et droites (« -ana », « -na », « -a »). L'ensemble constitue un treillis de toutes les unités correspondantes. Sur la base du coût cible, seules les n meilleures unités de chaque sous-séquence sont conservées. Le coût cible consiste à comparer le contexte phonétique d'une unité à celui de la séquence correspondante. Par exemple, pour synthétiser « hana », une unité « -an- » dont les contextes seraient « h » et « a » sera préférée à une unité dont les contextes seraient « g » et « i ». Cette étape pré-sélectionne donc des unités non uniformes en longueur en tenant compte du contexte phonétique, mais n'emploie aucune information relative à leurs variations acoustiques.

Le coût de concaténation favorise les unités qui présentent les critères suivants, non pondérés :

1. Transition CV ou VV, afin de minimiser les discontinuités, plus audibles dans ce genre de transitions.
2. Longue unité : une longue unité minimise les lieux de concaténation.
3. Longue superposition : le principe de concaténation est de *superposer* les parties communes aux unités contiguës. Par exemple, pour obtenir « sakida », les unités « mura*sa*ki » et « hi*kida*shi » seront superposées sur le « ki ». Selon l'auteur, plus la zone de superposition est longue, et meilleur est le résultat.

Avant d'être concaténées, les unités sont étirées ou raccourcies pour correspondre aux durées définies par le modèle prosodique.

Analyse

La pré-sélection dans ce système utilise uniquement la séquence phonétique, sans aucun critère prosodique ou acoustique. Ceci n'est pas gênant en Japonais, langue dont la structure syllabique la plus fréquente est CV et dont les variations prosodiques sont fort limitées. Les solutions proposées ici ne sont par contre pas applicables telles quelles aux langues riches en variations syllabiques et prosodiques, comme les langues romanes et germaniques.

Un autre inconvénient de ce système est que son coût de concaténation n'inclut aucune véritable distance acoustique, alors que la concaténation implique du traitement du signal.

Un dernier inconvénient majeur de cette méthode est l'importance du traitement du signal qu'elle utilise : les unités sont étirées ou raccourcies, et ne sont pas simplement concaténées, mais superposées sur de longs segments.

9.1.2 Sélection acoustique

Les principaux systèmes acoustiques sont ceux de Black & Campbell (1995), Hunt & Black (1996), Taylor & Black (1997) et Balestri *et al.* (1999).

D'autres systèmes emploient également une sélection acoustique (Beutnagel *et al.* 1999a,b, Yi *et al.* 2000, Bulyko & Ostendorf 2001, Allauzen *et al.* 2004), mais proposent surtout des méthodes d'optimisation : Beutnagel *et al.* (1999a,b) et Allauzen *et al.* (2004) récupèrent la méthode de sélection de Black & Campbell (1995), tandis que Bulyko & Ostendorf (2001) récupèrent celle de Taylor & Black (1997). Ces systèmes seront de ce fait traités dans la Section 9.2.

9.1.2.1 L'unité de base

Ces systèmes concatènent les unités contiguës en un point, au lieu de les superposer comme le faisait Sagisaka. Cette politique, qui vise à minimiser le traitement du signal, implique par contre de veiller à limiter les discontinuités acoustiques entre unités contiguës.

Ceci explique sans doute que, au lieu de pré-sélectionner uniquement les unités les plus longues, ces systèmes préfèrent pré-sélectionner des unités de longueur constante, en laissant à la recherche du meilleur chemin le soin de choisir des unités contiguës, pour autant que cela n'entraîne pas de discontinuité flagrante. Ces systèmes veulent donc favoriser une certaine constance dans la qualité. Selon les systèmes, l'unité de base est soit le phone (Black & Campbell 1995, Hunt & Black 1996, Taylor & Black 1997) soit le demi-phone (Balestri *et al.* 1999, Beutnagel *et al.* 1999a,b).

Selon les auteurs qui utilisent le demi-phone, cette unité favorise de meilleures concaténations. Ceci est certainement vrai lorsque les demi-phones appartiennent au même phonème (par exemple, /a| · |a/). Par contre, les erreurs de concaténation sont identiques à celles observées sur le phone lorsque les demi-phones appartiennent à des phonèmes différents

(par exemple, |a/ · /b|), étant donné que la concaténation est réalisée sur une phase de coarticulation.

9.1.2.2 Les critères de pré-sélection

Deux systèmes (Black & Campbell 1995, Hunt & Black 1996) utilisent principalement des informations acoustiques, comme la F0, l'énergie et le spectre, lors de la pré-sélection des unités. Une information linguistique est malgré tout utilisée : le contexte phonétique du phone. Cependant, cette information permet seulement de favoriser les unités qui présentent un contexte coarticulatoire similaire. Or, ce critère sert surtout à minimiser les discontinuités acoustiques. La courbe prosodique est donc complètement décrite en termes acoustiques.

Taylor & Black (1997) rejettent les informations acoustiques sur les phones de contexte, et préfèrent rechercher le phone lui-même sur la base de son accentuation, de sa position dans la syllabe et de sa position dans le groupe syntaxique. Cependant, ces critères linguistiques n'interviennent pas dans la pondération du candidat, dont le poids est exclusivement déterminé par rapport à l'ensemble auquel il appartient. La méthode consiste en effet à répartir les occurrences d'un phone en ensembles selon un critère de similarité acoustique. Dans ce contexte, le poids d'une unité est sa distance par rapport au *centroïde*, c'est-à-dire au centre théorique de l'ensemble. Cette distance acoustique contraint le système à choisir les unités les plus neutres possibles.

Seul le système de Balestri *et al.* (1999) pré-sélectionne sur des critères linguistiques pondérés : l'adéquation d'un demi-phone est déterminée à l'aide d'une fenêtre-cloche, centrée sur le demi-phone et délimitée par le premier demi-phone différent. Sur cette fenêtre est calculé un degré de similarité linguistique en termes de classe articulatoire, d'accentuation, et de voisement. La forme de la fenêtre permet d'accorder plus d'importance aux phones situés au centre de la fenêtre. Cependant, la méthode dans sa globalité ne peut être considérée comme linguistique, parce que les demi-phones sont repondérés en fonction des valeurs de F0 et de durée attendues par le système. En outre, lorsque la meilleure suite d'unités a été sélectionnée, toute unité s'écartant de la cible au niveau de la F0 ou de la durée subit un lourd traitement du signal, ce qui va à l'encontre de l'objectif initial.

9.1.2.3 Les critères de concaténation

Certains systèmes calculent une distance cepstrale (Black & Campbell 1995, Hunt & Black 1996, Taylor & Black 1997) à laquelle les différences de F0 et d'énergie peuvent s'ajouter (Hunt & Black 1996). Balestri *et al.* (1999) se basent par contre sur la F0 et la différence de durée entre demi-phones à concaténer.

Le lieu de concaténation dépend du système. Deux systèmes (Black & Campbell 1995, Taylor & Black 1997) calculent le meilleur point de concaténation entre deux unités contiguës, et retournent pour ce point le poids de concaténation. Ainsi, le point de conca-

ténation d'une unité varie en fonction de l'unité avec laquelle elle est concaténée. Les deux autres (Hunt & Black 1996, Balestri *et al.* 1999) calculent par contre le coût de concaténation en frontière d'unité. Le point de concaténation est dans ce cas toujours le même.

9.1.2.4 Pondération

Black & Campbell (1995) ont testé une estimation semi-automatique des poids du système. Nous qualifions cette estimation de semi-automatique parce qu'elle consiste à définir *manuellement* un ensemble de pondérations et de choisir ensuite *automatiquement* la meilleure pondération de cet ensemble. L'estimation utilise un ensemble de phrases n'appartenant pas au corpus de sélection. Chaque phrase est synthétisée autant de fois qu'il y a de pondérations, et chaque résultat est aligné avec l'original par calcul de leur distance cepstrale euclidienne. Cette distance caractérise l'adéquation de la pondération utilisée. La pondération ayant la meilleure distance sur l'ensemble des phrases testées est retenue. Comme l'estiment eux-mêmes les auteurs, la méthode accorde malheureusement plus d'importance à la courbe générale du signal qu'aux problèmes de concaténation parfois francs arrivant aux jointures, alors que l'auditeur humain, lui, est choqué par ces problèmes de concaténation.

Chez Hunt & Black (1996), les pondérations du coût cible et du coût de concaténation sont réalisées séparément. Concernant le coût de concaténation, les auteurs préconisent une simple combinaison linéaire des critères utilisés. Concernant le coût cible, la méthode permet de déterminer des pondérations différentes en fonction du phonème. Le principe est de considérer chaque occurrence d'un phonème comme une cible. Pour cette cible, les 20 unités les plus proches en terme de distance cepstrale euclidienne sont sélectionnées. Les pondérations des critères du coût cible sont alors déterminées en estimant ces distances par régression linéaire multiple (Draper & Smith 1998). Les auteurs estiment que les résultats en terme de synthèse sont malheureusement fort comparables à ceux obtenus avec la méthode de pondération de Black & Campbell (1995). L'avantage, par contre, est que les poids sont estimés automatiquement. L'apparente contre-performance de la méthode de pondération ne vient probablement pas de la méthode en elle-même, mais du point de comparaison choisi : la distance cepstrale entre les unités n'est certainement pas une information suffisamment discriminante pour pondérer de manière efficace des critères comme la durée et les caractéristiques articulatoires du contexte.

Nous avons déjà mentionné que les critères de recherche de la cible, chez Taylor & Black (1997), ne sont pas pondérés parce que le coût d'une unité est sa distance par rapport au centroïde de l'ensemble auquel elle appartient. Un poids prépondérant est par contre accordé au coût de concaténation.

Chez Balestri *et al.* (1999), le coût de concaténation est également favorisé, en sur-pondérant manuellement les unités adjacentes et les faibles coûts de concaténation.

9.1.2.5 Analyse

Ces différents systèmes génèrent de la parole de synthèse de haute qualité, dont certains échantillons sont presque indifférentiables de la parole humaine. Malheureusement, quelles que soient leurs particularités, ces systèmes souffrent tous du même mal : une trop grande rigidité de la prosodie, du fait d'une pré-sélection guidée à l'aide de critères acoustiques, comme la F0 ou la durée.

Une raison simple explique l'insuffisance de ces critères acoustiques : les valeurs de ces critères sont générées par des modèles prosodiques, à partir d'informations linguistiques comme la structure syllabique, l'accentuation de la syllabe, les phonèmes qui constituent la syllabe ou la position de la syllabe par rapport à la pause. Or, les modèles prosodiques présentent trois inconvénients majeurs :

1. Qu'ils soient réalisés par des experts ou appris sur des corpus, ces modèles sont généralement fort réducteurs : ils ne tiennent compte que des structures linguistiques les plus représentatives ou les plus fréquentes, ce que l'on pourrait appeler la *norme* prosodique. Toute phrase qui s'en écarte sera dès lors malgré tout modélisée selon cette norme.
2. Pour un phrase donnée à laquelle correspond une seule analyse linguistique, l'être humain peut réaliser des courbes prosodiques fort différentes sans pour autant marquer dans ces courbes d'informations sémantiquement pertinentes. Les modèles prosodiques sont par contre déterministes par nature : pour une liste de valeurs linguistiques données, un modèle donné génère toujours les mêmes valeurs acoustiques. Ceci signifie que les systèmes de synthèse dirigés par ce type de modèles répètent inlassablement les mêmes patrons prosodiques de phrase en phrase. Ceci risque d'installer une certaine monotonie à l'écoute, dont une conséquence possible est le désintérêt de l'auditeur pour la parole de synthèse.
3. Un modèle prosodique est par nature dépendant de la langue. L'adaptation du système de synthèse à une nouvelle langue demande donc le développement d'un nouveau modèle prosodique complet.

Historiquement, les modèles prosodiques étaient une nécessité en synthèse. En effet, les systèmes par règles ainsi que les systèmes par concaténation d'unités neutres devaient connaître la courbe prosodique à *réaliser*. L'introduction d'unités non uniformes a dégagé la synthèse de cette obligation, mais les premiers auteurs, fidèles à l'esprit des systèmes établis, ont gardé cette habitude probablement... réductrice.

Forts de ce constat, certains chercheurs ont décidé d'abandonner les informations prosodiques de type acoustique au profit d'informations purement linguistiques.

9.1.3 Sélection linguistique

Les systèmes libérés de tout modèle prosodique sont ceux de Breen & Jackson (1998a,b), Taylor & Black (1999), Taylor (2000) et Prudon & d'Alessandro (2001). Etant

donné le peu de caractéristiques communes partagées par ces systèmes, ceux-ci sont présentés séparément.

9.1.3.1 Breen & Jackson (1998a,b)

Elaborée dans le contexte de la téléphonie, cette méthode est motivée par le désir de maximiser l'intelligibilité, fût-ce au détriment du naturel de la voix.

Les auteurs ne considèrent pas les phones, mais restent au niveau des phonèmes : leurs transcriptions sont donc du niveau du phonème, évitant de modéliser les effets de la coarticulation, comme l'amuïssement, la nasalisation, l'assourdissement ou la sonorisation.

Afin de maximiser l'intelligibilité, les auteurs ne font aucun élagage parmi les unités de la pré-sélection. Lors de la recherche d'un phonème A appartenant à une séquence phonologique $FDBACEG$, le corpus est interrogé avec le triphonème BAC. Cette recherche retourne toutes les occurrences du triphonème lui-même s'il existe, mais également celles des triphonèmes qui divergent à droite ($BA\_$), à gauche ($\_AC$) ou des deux côtés ($\_A\_$). A des fins de concaténation, il est dès lors nécessaire de définir une distance.

La distance choisie est linguistique : chaque phonème est décrit à l'aide d'informations segmentales et suprasegmentales. Les informations segmentales retenues sont de nombreuses caractéristiques articulatoires comme, par exemple, le mode et le lieu d'articulation, la labialité ou la nasalité. Les informations suprasegmentales sont uniquement l'accentuation et la place du phonème dans la syllabe (onset, nucleus, coda). De la sorte, une distance peut être déterminée pour deux triphonèmes qui divergent sur le phonème de concaténation, comme $BA\underline{C}$ et $\underline{H}EG$.

En cas de divergence segmentale, les unités peuvent au plus être juxtaposées. En cas de simple divergence suprasegmentale, les unités peuvent être *superposées*. Le coût de concaténation, constitué exclusivement de critères linguistiques, favorise donc les divergences suprasegmentales aux divergences segmentales, et la superposition à la juxtaposition.

Constat. Ce système a été optimisé pour la téléphonie. Comme le signalent les auteurs, l'application peut dès lors utiliser de nombreuses ressources et ne doit pas impérativement être temps réel. Ceci explique que la méthode utilise tous les candidats du corpus de parole et ne réalise aucun élagage avant la recherche du meilleur chemin. En outre, dans le contexte de la téléphonie, l'objectif des auteurs est de faire primer l'intelligibilité au risque de quelques discontinuités acoustiques. Selon leur analyse, l'objectif est atteint.

Cependant, la superposition d'unités, aussi similaires soient-elles, demande un important traitement du signal qui diminue le naturel de la parole.

En l'absence de critères acoustiques, on peut enfin s'interroger sur la pertinence des critères linguistiques choisis : guident-ils la courbe prosodique de la parole de synthèse ? La réponse est non. Parmi les informations retenues, seuls les critères suprasegmentaux sont des indices prosodiques. Ils ne sont cependant pas suffisants : aucune information ne décrit la structure syllabique et aucun indice ne situe l'unité par rapport à la pause. Ces éléments

sont pourtant des facteurs influant sur la durée des sons et sur l'évolution de la courbe intonative. Il n'y a donc aucun véritable guide prosodique dans ce système.

9.1.3.2 Taylor & Black (1997)

Les auteurs disent se situer au niveau phonologique, poussant d'ailleurs le concept un peu plus loin que Breen & Jackson (1998a,b) : dans le corpus anglais qu'ils utilisent, même les phénomènes de contraction (par exemple « gonna ») sont étiquetés avec leur forme canonique (« going to »). Cependant, contre toute attente, la courbe acoustique est malgré tout modélisée, et toute unité qui s'en écarte est fortement sanctionnée au moment de la synthèse.

La longueur des unités recherchées par le système va du syntagme au phonème. Cependant, le système favorise les longues unités, ne recensant les unités plus courtes que si les unités longues n'appartiennent pas au corpus. Les candidats trouvés sont uniquement pondérés selon des critères phonologiques et suprasegmentaux. Ces critères sont pondérés manuellement, ce que les auteurs justifient par le petit nombre de critères.

Aucune distance acoustique n'est utilisée lors de la recherche de la meilleure suite d'unités. L'acoustique resurgit par contre au moment de l'étape de synthèse. En effet, la F0 et la durée de chaque unité sont comparées aux valeurs pré-calculées par le système. Si elles s'en écartent au-delà d'un seuil posé, un traitement du signal est appliqué à l'unité.

D'après les auteurs, ce système présente l'avantage de très bien modéliser le corpus de parole, de sorte qu'une phrase du corpus est toujours choisie lorsqu'il s'agit de la phrase à synthétiser. Les auteurs estiment dès lors que le système est particulièrement adapté à la synthèse de phrases d'un domaine particulier, pour autant que ces phrases soient intégrées au corpus.

Constat. L'avantage souligné par les auteurs n'est pas propre à leur système. En somme, pour autant que le corpus et la phrase à synthétiser aient été analysés avec le même outil, il est normal et même rassurant que le système modélise correctement le corpus. Le contraire mettrait au jour une erreur de l'algorithme de sélection : une phrase du corpus qui est proposée à la synthèse *doit* être choisie par la sélection.

Cette approche présente en outre trois inconvénients majeurs :

1. Il est dangereux d'étiqueter phonologiquement un corpus de parole. Cet abus ne sera pas perceptible si le système sélectionne un syntagme ou un mot, mais introduira inévitablement des erreurs si le système doit sélectionner des unités plus petites. Prenons un exemple : s'il faut synthétiser « going down » et que le système doit descendre au niveau de la syllabe « ing », le risque est de sélectionner une partie de la prononciation « gonna » étiquetée à tort « going to ».
2. La sélection exclusive d'unités longues hypothèque les possibilités d'obtenir une courbe prosodique sans discontinuité : le système est contraint de concaténer des unités

longues, dont les frontières peuvent être acoustiquement incompatibles. Il n'est certainement pas inutile de laisser à la sélection le soin d'estimer le meilleur compromis entre longueur et qualité globale.

3. La sélection linguistique ne présente aucun intérêt si tout « écart prosodique » est corrigé par le module de synthèse. D'une part, le traitement du signal risque de dégrader la qualité du signal. D'autre part, ces corrections forcent la synthèse à répéter inlassablement le même patron prosodique, alors que l'idée de la sélection linguistique est initialement d'éviter cet écueil.

9.1.3.3 Prudon & d'Alessandro (2001)

Au contraire des systèmes linguistiques précédents, les auteurs reviennent à la sélection d'une unité de base, en l'occurrence le phone.

La pré-sélection est exclusivement linguistique. Le phone cible est recherché avec le bon phonème contextuel droit. Les critères de pré-sélection sont :

- La position du phone dans le mot : début, intérieur, fin ;
- Le type de la syllabe.

Peu de critères sont donc utilisés, ce que les auteurs justifient par la difficulté de comprendre le sens d'un critère lorsque de nombreux critères sont employés. Les deux critères choisis reçoivent manuellement le même poids (0.5).

Le coût de concaténation de deux unités s'établit quant à lui exclusivement sur :

- L'adjacence des unités dans le corpus : les unités contiguës sont préférées aux unités non contiguës ;
- La différence en terme de F0, dans le cas d'unités non contiguës.

Les deux critères sont manuellement pondérés. La différence de F0 intervient pour peu dans le coût (0.15), au profit de l'adjacence des unités (0.85).

Lors de la recherche du meilleur chemin, le coût cible est légèrement favorisé (0.6) par rapport au coût de concaténation (0.4) [2].

Constat. Les échantillons de parole proposés par les auteurs confirment leur analyse : le système n'évite pas de fréquentes discontinuités acoustiques, et choisit souvent des unités dont l'accentuation est inappropriée.

Les discontinuités acoustiques sont probablement dues au manque de finesse du coût de concaténation : la distance acoustique entre deux unités de parole ne peut se réduire à une simple différence de F0. Afin de tenir compte de toute la richesse de la parole, d'autres mesures doivent accompagner la F0, comme l'énergie ou le spectre.

[2] Notons que dans leur article, les auteurs disent encourager le coût cible, mais présentent un tableau dans lequel le coût cible reçoit 0.4 et le coût de concaténation, 0.6. Nous avons donc inversé ces valeurs.

Les erreurs d'accentuation ont probablement deux origines. La première est peut-être une mauvaise couverture des unités nécessaires au système en fonction des critères utilisés. La seconde est certainement le manque d'informations prosodiques du coût cible :

- La position dans le mot est certainement un critère pertinent en français, mais insuffisant. L'accentuation dépend aussi, voire surtout, de la position du mot dans le groupe de souffle.
- Le type de la syllabe influe certainement sur la durée des sons qui la constituent, mais cette information devrait être complétée par la position du phone recherché dans la syllabe : par exemple, la recherche d'une consonne appartenant à une syllabe de type CVC ne devrait pas donner le même résultat selon que la consonne appartient à l'onset ou au coda.

9.1.3.4 Analyse

En refusant les modèles prosodiques, les systèmes de Breen & Jackson (1998a,b) et Taylor & Black (1997) n'ont pas simplement supprimé le recours aux informations acoustiques dans la phase de pré-sélection : ils ont supprimé *toute* information prosodique, fût-elle linguistique. A ceci s'ajoute le fait que Breen & Jackson (1998a,b) suppriment toute distance acoustique du coût de concaténation, tandis que Taylor & Black (1997) pré-sélectionnent exclusivement les unités les plus longues, au détriment certain de la continuité acoustique. Pourtant, le fait de manipuler des unités *non uniformes*, qui présentent à dessein des variations acoustiques, impose de choisir les unités à l'aide de critères prosodiques et de vérifier la qualité de leur continuité acoustique à l'aide de critères... acoustiques. Ce rejet complet de l'acoustique est donc étonnant.

Dans la répartition entre informations linguistiques et mesures acoustiques, le système de Prudon & d'Alessandro (2001) est certainement plus équilibré : les informations linguistiques guident la pré-sélection des unités, tandis que les mesures acoustiques vérifient la qualité des transitions acoustiques entre unités. Le système, dans ce cas-ci, pèche par contre par manque d'informations : la pré-sélection manque d'indices suprasegmentaux, et la F0 ne suffit pas à estimer la véritable distance acoustique entre deux unités.

9.2 Optimisation de la recherche

Nous avons mentionné que le processus de recherche se doit de rester temps réel (cf. Section 8.5.2.4). Pour ce faire, la plupart des systèmes réduisent les occurrences trouvées par la pré-sélection aux n meilleurs candidats (entre 10 et 50) en fonction du coût cible. Black & Campbell (1995) effectuent en outre une seconde réduction : pour chaque candidat, ils ne conservent dans le treillis que les m meilleures concaténations (entre 20 et 50). La recherche du meilleur chemin est ensuite réalisée sur ce treillis élagué.

Outre cet élagage, plusieurs systèmes ont recours à des structures de données qui tâchent d'optimiser la pré-sélection, le calcul du coût de concaténation ou la totalité du processus de sélection.

9.2.1 Optimisation de la pré-sélection

Les systèmes qui optimisent la phase de pré-sélection ont systématiquement recours à un arbre, qu'il s'agisse d'un arbre de décision, d'un arbre phonologique ou d'un arbre linguistique. Nous présentons ces structures avant d'en proposer une analyse.

9.2.1.1 Arbre de décision

Chez Taylor & Black (1997), la recherche des unités candidates est réalisée dans un arbre de décision.

Un arbre de décision (Breiman *et al.* 1984, Quinlan 1993, van den Bosch 1997) est une représentation arborescente qui permet de prendre une décision en posant un certain nombre de questions dans un ordre optimisé par un apprentissage. L'apprentissage détermine l'importance des questions et donc l'ordre optimal dans lequel ces questions doivent être posées. Les niveaux de l'arbre de décision sont construits en respectant cet ordre, la question la plus pertinente correspondant au premier niveau de l'arbre.

Lors du questionnement de l'arbre, les réponses aux questions sont pré-établies. L'arbre est parcouru selon l'ordre d'importance des questions, et le questionnement ne s'interrompt que lorsqu'une feuille de l'arbre est atteinte ou que, au niveau atteint, aucune branche ne présente la réponse attendue pour la question posée. Lorsque la recherche s'interrompt, l'arbre renvoie la décision *par défaut* du niveau atteint, qui est la décision la plus probable retenue par l'apprentissage.

L'arbre construit ici est un CART, abréviation de *Classification And Regression Tree*. La particularité de ce type d'arbre est qu'il ordonne les questions de l'arbre selon un *critère de pureté* donné. A un niveau de l'arbre, l'algorithme teste toutes les questions possibles, et divise la partition selon la question qui maximise le critère de pureté. Le lecteur intéressé par le détail de la construction d'un CART consultera utilement (Breiman *et al.* 1984).

Dans le système de Taylor & Black (1997), un CART est construit pour chaque phone :

- Le critère de pureté est la distance acoustique entre les unités de la partition.
- Les questions du partitionnement sont les critères de la cible.
- A l'aide de ce critère de pureté et des questions de la cible, l'ensemble d'occurrences acoustiques du phone est donc divisé en partitions d'occurrences acoustiquement proches.

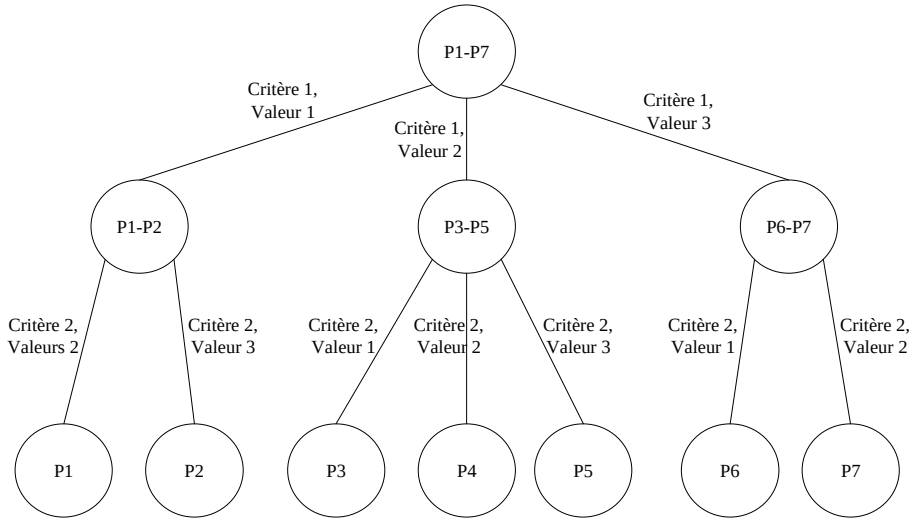


FIG. 9.1: CART où chaque nœud final contient une partition P_i de réalisations acoustiques similaires Taylor & Black (1997)

Au cours de la pré-sélection, l'arbre d'un phone est interrogé à l'aide des critères de la cible, et la partition correspondante est retournée par l'arbre (cf. Figure 9.1).

9.2.1.2 Arbre phonologique

Chez Breen & Jackson (1998a), le corpus de parole est représenté sous la forme d'un arbre phonologique contextuel. Dans l'arbre (cf. Figure 9.2),

- le premier niveau contient les phonèmes cibles,
- le second niveau contient, pour chaque phonème, les contextes phonétiques immédiats ({phonème gauche, phonème droit}) rencontrés dans le corpus,
- le troisième niveau recense l'ensemble des occurrences pour un phonème selon un contexte immédiat donné.

L'unité maximale représentée par cet arbre est donc le triphonème : trois *phonèmes* complets [3]. Le principe pourrait cependant être étendu, au détriment de la charge de calcul.

9.2.1.3 Arbre linguistique

Chez Taylor & Black (1999), le corpus est représenté sous la forme d'un arbre linguistique par phrase. Comme le montre la Figure 9.3, les informations se répartissent entre les niveaux syntagmatiques, lexicaux, syllabiques et phonologiques. Tous les niveaux, sauf le dernier, indiquent en outre l'accentuation globale de l'unité, « faible » ou « forte ».

[3] Les auteurs emploient à tort le terme *triphone* : leur unité ne va pas de partie stable en partie stable, mais inclut les phases de coarticulation.

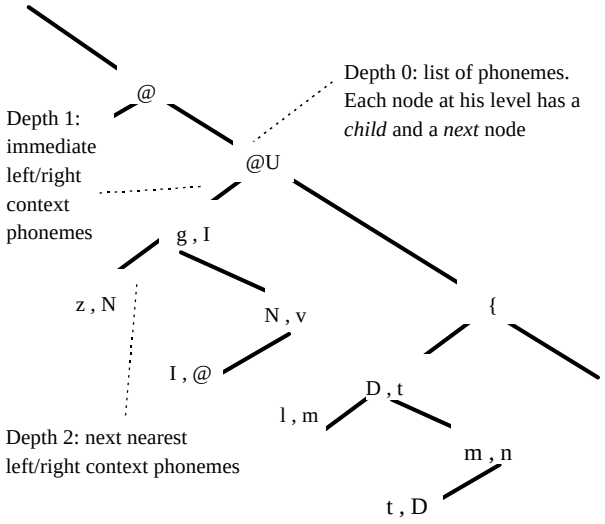


FIG. 9.2: Arbre phonologique contextuel. Figure reprise de (Breen & Jackson 1998a). La Figure présente 2 niveaux de contexte, mais dans l'implémentation décrite par les auteurs, seul 1 niveau de contexte est utilisé

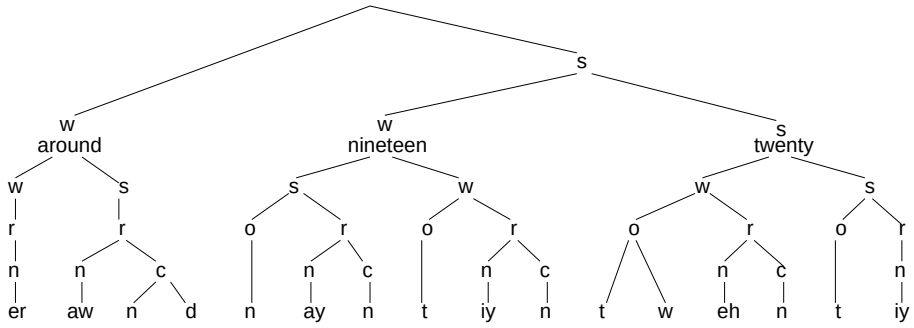


FIG. 9.3: Arbre linguistique. Figure reprise de (Taylor & Black 1999). Dans l'arbre, « w » note les nœuds non accentués (*weak*) et « s », les nœuds accentués (*strong*).

Lors de la pré-sélection, la phrase à synthétiser est représentée sous la même forme arborescente. Le principe de la pré-sélection est de rechercher l'unité la plus longue possible au niveau le plus haut des arbres. S'il y a un résultat à ce niveau, la recherche est interrompue et passe à la partie de la phrase qui n'est pas encore traitée. S'il n'y a pas de résultat, le système descend d'un niveau dans l'arbre et recommence la recherche.

9.2.1.4 Analyse

Un arbre permet de retrouver rapidement un ensemble de candidats, mais ne les pondère pas : un arbre de décision ne retient pas la pondération utilisée pour répartir les critères entre ses niveaux, et les arbres linguistiques ou phonologiques sont construits sans tenir compte de la pondération. La pré-sélection est donc accélérée, mais n'inclut pas le calcul du coût cible.

L'optimisation de la phase de pré-sélection n'est en outre pas suffisante pour rendre le système efficace. Comme le soulignent Beutnagel *et al.* (1999b), les coûts de concaténation prennent à eux seuls près de 80% du temps de sélection et un peu plus de 50% du temps total du processus de synthèse.

9.2.2 Optimisation des coûts de concaténation

Optimiser les coûts de concaténation consiste à les pré-calculer et à les mémoriser. Les auteurs constatent cependant qu'il n'est pas réaliste de mémoriser la totalité des coûts de concaténation : Beutnagel *et al.* (1999a), par exemple, expliquent que leur corpus, fort de 84 000 unités, autorise 1,8 milliards de concaténation. Les structures proposées pour cette optimisation sont une table de hashage et un transducteur pondéré.

9.2.2.1 Table de hashage

Afin de choisir les concaténations à conserver, Beutnagel *et al.* (1999a) ont synthétisé un corpus de 250 000 phrases avec leur système. Ce corpus présente 1,2 millions de concaténations distinctes, qui ont été retenues et mémorisées dans une table de hashage.

Une table de hashage (Aho *et al.* 1974, 1986) est une table dont les éléments sont indexés à l'aide d'une fonction de hashage : la fonction prend en entrée une clef et retourne en sortie l'indice de l'élément recherché dans la table. Dans une table de hashage, il arrive que plusieurs éléments soient mémorisés dans la même case. On parle de *collision*. Trop de collisions sont le résultat d'une mauvaise fonction de hashage. La fonction de hashage doit donc être adaptée aux valeurs de la clef.

Dans le système de Beutnagel *et al.* (1999a), la clef est le couple d'unités à concaténer. L'indice retourné est la position du coût de concaténation dans le tableau. L'optimisation de la fonction est quant à elle basée sur les travaux de Tarjan & Yao (1979). En cours de sélection, lorsqu'un coût n'appartient pas à la table, une valeur par défaut et très élevée est attribuée au coût de concaténation des unités concernées.

L'inconvénient de l'approche est qu'elle se base sur un corpus de phrases synthétisées : en fonction de la fréquence des unités dans la langue, certaines unités seront surreprésentées dans le corpus, alors que d'autres en seront absentes. Cette approche risque donc d'entraîner un déséquilibre dans le coût de concaténation au profit des unités fréquentes.

9.2.2.2 Transducteur pondéré

Chez Yi *et al.* (2000), le coût de concaténation est représenté sous la forme d'un transducteur pondéré. A priori, ce transducteur devrait compter un état par unité, et une transition entre tout couple d'états correspondant à un couple d'unités concaténables, à moins qu'un élagage ne soit réalisé. Cependant, afin d'éviter cet élagage, les auteurs ont préféré calculer les coûts de concaténation à l'aide de *critères articulatoires*. Comme le montre la Figure 9.4, ces critères sont représentés dans la machine à l'aide d'états intermédiaires placés entre les états (unités) à concaténer. Ces états intermédiaires permettent de réduire

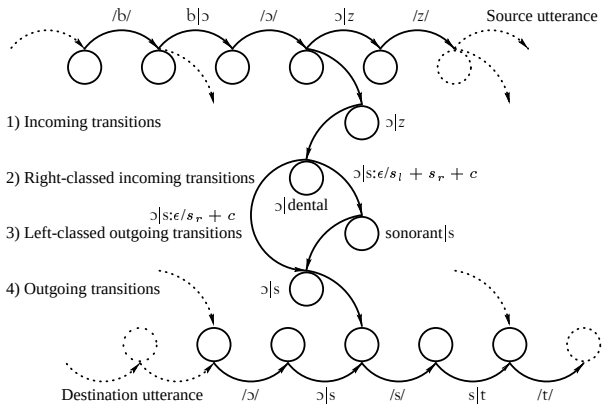


FIG. 9.4: Transducteur phonologique. Figure reprise de (Yi *et al.* 2000)

considérablement le nombre de transitions de la machine.

Malheureusement, cette méthode de calcul du coût de concaténation ne peut éviter certaines concaténations fort abruptes, étant donné qu'aucun critère acoustique n'est utilisé. Les auteurs envisagent d'ailleurs de modifier les segments de parole au niveau de la F0 et de la durée à l'aide de traitement du signal. Cette optimisation du système dégrade donc la qualité de la synthèse obtenue.

9.2.3 Optimisation du processus global

Lorsque l'ensemble du processus de sélection est optimisé, une seule structure de données représente les deux phases de la sélection.

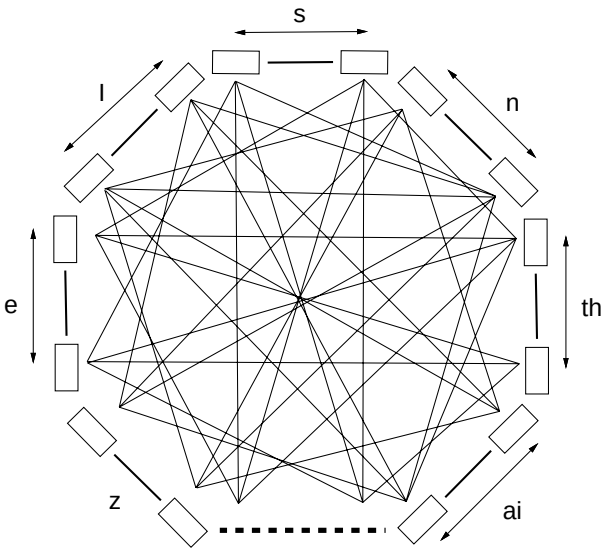


FIG. 9.5: Réseau d'états. Figure reprise de (Hunt & Black 1996)

9.2.3.1 Réseau d'états

Chez Hunt & Black (1996), le corpus de parole est représenté sous une forme fort similaire à une machine à états finis : un réseau d'états (cf. Figure 9.5), où chaque état a un poids, le coût cible, et où toute transition entre deux états est pondérée par le coût de concaténation.

Un point n'est cependant pas éclairci par les auteurs : *a priori*, rien n'empêche la sélection de choisir plusieurs fois la même unité à différentes positions d'une même phrase. Or, si le coût cible est contenu dans l'état de l'unité, il semble difficile que le même état puisse représenter plusieurs fois la même unité avec des coûts cibles différents. La logique voudrait donc que le coût cible soit unique pour une unité donnée, comme la distance par rapport au centroïde dans les partitions de Taylor & Black (1997). Or, le coût cible de Hunt & Black (1996) est basé sur la F0, l'énergie et la durée.

En outre, comme nous l'avons signalé, un réseau d'états qui représenterait l'ensemble des coûts de concaténation possibles dans un corpus de plusieurs milliers d'unités aurait une taille considérable. Cependant, les auteurs n'indiquent pas la taille du corpus utilisé et ne précisent pas si leur réseau est le résultat d'un élagage.

9.2.3.2 Transducteur pondéré

Comme le montre la Figure 9.6, le système de Bulyko & Ostendorf (2001) rassemble en un seul transducteur pondéré à la fois le coût cible et le coût de concaténation.

Une partie des états du transducteur (partie supérieure de la Figure) sont les unités du corpus. Chacun de ces états possède une transition pondérée par le coût cible. Comme

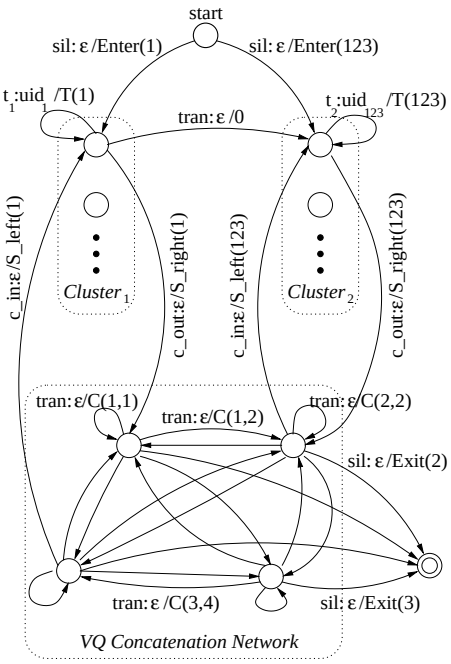


FIG. 9.6: Transducteur acoustique. Figure reprise de (Bulyko & Ostendorf 2001)

dans le système de Hunt & Black (1996), le coût cible ne peut dès lors valoir que la distance de l'unité par rapport au centroïde de la partition à laquelle elle appartient.

Les autres états du transducteur (partie inférieure de la Figure) représentent le coût de concaténation : ils correspondent à des ensembles de vecteurs acoustiques similaires. Chaque état représente un ensemble de vecteurs acoustiques, prélevés aux frontières des unités, et jugés similaires par quantification vectorielle (Linde *et al.* 1980, Gray 1984). Ceci permet de proposer un seul coût de concaténation pour plusieurs frontières d'unités. Au niveau des coûts de concaténation, n unités sont donc compressées en m états, avec $m \ll n$.

Cette compression nécessite cependant de prévoir à chaque état représentant un ensemble de vecteurs acoustiques, autant de transitions de concaténation qu'il y a d'ensembles de vecteurs acoustiques. Ce sont les transitions étiquetées *tran*. Le transducteur est donc *non déterministe*, ce qui implique que le temps de parcours de la machine n'est pas linéairement proportionnel à la taille de la phrase à synthétiser (cf. Section 2.3.2).

En se basant sur la quantification vectorielle, la distance entre deux unités n'est en outre plus qu'une approximation, puisque elle n'est plus estimée directement sur les unités.

9.2.3.3 Modèle de langue par transducteur pondéré

Chez Allauzen *et al.* (2004), un modèle de langue remplace la résolution classique du coût cible et du coût de concaténation.

L'idée est d'estimer la probabilité qu'une suite d'unités ($U = u_1 \cdots u_n$) corresponde

à une suite de cibles ($C = c_1 \cdots c_n$) :

$$U = \arg\max_U P(U|C) \tag{9.2.3.1}$$

ce qui, par le théorème de Bayes (Boîte *et al.* 2000), se résout classiquement par :

$$U = \arg\max_U \frac{P(C|U)P(U)}{P(C)} \tag{9.2.3.2}$$

dont le dénominateur, $P(C)$ est considéré comme constant et donc supprimé.

Afin de pouvoir estimer la probabilité *a priori* ($P(U)$) et la probabilité *a posteriori* ($P(C|U)$) du modèle, les auteurs ont eu besoin d'un corpus de phrases de synthèse. Le corpus qu'ils ont utilisé a été généré à partir du système de sélection de Beutnagel *et al.* (1999a), et représente environ 8 millions de séquences d'unités. Il a été étiqueté en cibles et en unités et les parties du modèle ont été entraînées.

En cours de synthèse, lors de l'étape de sélection, un automate A_C est construit à partir de la suite de cibles C, et est composé avec le modèle : $R = A_C \circ P(C|U) \circ P(U)$. La recherche du meilleur chemin est réalisée sur le résultat R de ces compositions. La nouvelle sélection, basée sur ce modèle statistique, est en moyenne 2,6 fois plus rapide que la sélection d'origine.

Ce système présente cependant deux inconvénients :

1. Les auteurs reconnaissent eux-mêmes que s'il est plus rapide, le système ne modélise au mieux que les coûts cibles et coûts de concaténation du système d'origine.
2. Les modèles de langue sont fortement influencés par le choix des phrases qui constituent le corpus d'entraînement. Or, cette influence est difficilement estimable, ce qui complique tout processus d'ajustement du système en fonction d'erreurs de synthèse. Il semble dès lors préférable de conserver les pondérations du système d'origine, en utilisant une autre méthode d'optimisation.

9.3 Limites des systèmes de l'état de l'art

Les premiers systèmes de l'état de l'art en synthèse par sélection d'unités non uniformes, probablement dans la continuité de la synthèse par concaténation, se sont construits autour d'une majorité d'informations acoustiques. Cependant, l'analyse des résultats obtenus par ces systèmes a mis au jour les deux inconvénients liés à l'utilisation de valeurs acoustiques pour la pré-sélection : le processus de synthèse doit inclure un modèle prosodique dépendant de la langue, et la prosodie générée, fort neutre et répétitive, a tendance à diminuer l'agrément de l'auditeur.

Forts de ce constat, certains chercheurs ont développé leurs systèmes de sélection autour d'informations presque exclusivement linguistiques. Cependant, ces systèmes ont commis deux excès qui nuisent au naturel et à la qualité de leur synthèse. D'une part,

les critères linguistiques choisis comprennent très peu de critères suprasegmentaux, ce qui nuit à la définition de la courbe prosodique. D'autre part, les mesures acoustiques ont été réduites au strict minimum, voire supprimées du système, ce qui nuit à la continuité des unités.

Quels que soient les critères de sélection utilisés, des méthodes de pondération des critères de la sélection ont été tentées. Automatiques ou semi-automatiques, ces méthodes sont toutes acoustiques : elles estiment une distance entre un candidat et un référent, que le processus se situe au niveau de l'unité ou de la phrase. Ces méthodes ont cependant deux inconvénients : d'une part, elles ne peuvent tenir compte de critères linguistiques de sélection, d'autre part, elles favorisent systématiquement le coût de concaténation au détriment du coût cible. Il faut noter que les chercheurs estiment ce choix pertinent, étant donné que l'oreille humaine est plus choquée par les discontinuités acoustiques que par les erreurs prosodiques. Une analyse linguistique du problème aurait pourtant tendance à dire que le système favorise à ce point le coût de concaténation parce que les critères du coût cible ne sont pas représentatifs de la prosodie de la langue.

De nombreux systèmes ont en outre eu recours à des mécanismes d'optimisation. Certains n'ont optimisé que la phase de pré-sélection au travers de structures arborescentes. Cette structure s'est avérée insuffisante parce qu'elle ne pondère pas automatiquement les critères du coût cible.

D'autres systèmes ont uniquement optimisé la phase de sélection, soulignant l'importance du pré-calcul des coûts de concaténation et la nécessité d'un élagage. Un système a cependant tâché de conserver la totalité des distances, mais au détriment de la qualité du coût qui n'inclut de ce fait plus d'information acoustique.

Les derniers systèmes ont tâché d'optimiser la totalité du processus. Certains d'entre eux ont créé un réseau d'états ou un transducteur unique, limitant de ce fait le coût cible à une distance avec le centroïde de la partition. Un dernier système a proposé de représenter sous la forme de transducteurs un modèle de langue entraîné sur un corpus de phrases générées par un autre système, mais au risque de sur-entraîner le modèle en fonction des phrases du corpus, dont l'influence sur la qualité des résultats est difficilement estimable.

Trois points faibles récurrents émanent donc de l'état de l'art. Le plus important est probablement le choix des critères de sélection. Le second, étroitement dépendant des critères de sélection, est le mode de pondération de ces critères. Le dernier est l'influence de la méthode d'optimisation sur le comportement général du système. C'est sur cette analyse que nous avons dégagé les principes du nouveau système de sélection que nous proposons, baptisé LiONS.

Chapitre 10

LiONS

LiONS [1] est l'abréviation de « *Linguistically Oriented Non-uniform units Selection* ». Il s'agit d'un système de sélection d'unités non uniformes *orienté* linguistique et non *exclusivement* linguistique.

Ce système a d'abord été le fruit d'une collaboration dont les résultats ont été présentés dans (Colotte & Beaufort 2004, 2005) et qui ont donné lieu au brevet décrit dans (Beaufort & Colotte 2006). Dans cette première version de LiONS, l'attention a été concentrée sur les critères à retenir pour la sélection et sur la méthode de pondération à utiliser, sans considération pour l'optimisation. L'objectif était principalement de faire une preuve de concept, que nous détaillons en Section 10.1.

L'optimisation de la sélection, résultat d'un travail strictement personnel, a été réalisée dans un second temps à l'aide de machines à états finis. Comme nous le montrerons, cette optimisation a également été l'occasion de reconsidérer l'éventail des critères retenus pour la sélection et de reconsidérer en partie la méthode de pondération. Optimisation et révisions font l'objet de la Section 10.2.

10.1 Preuve de concept

Cette section s'ouvre par la présentation du postulat et des hypothèses sur lesquels se fonde globalement le système LiONS. Elle présente ensuite succinctement le synthétiseur dans lequel LiONS a été intégré, et plus particulièrement son module de traitement de la langue, qui génère les informations linguistiques utilisées par LiONS.

Sur cette base, nous détaillons les critères retenus pour le coût cible et le coût de concaténation, et les raisons de ces choix. Nous décrivons ensuite le corpus de parole que nous avons utilisé et les différentes étapes de l'entraînement réalisé sur ce corpus. La dernière phase de l'entraînement, la pondération des critères, fait l'objet d'un point complet au vu de l'importance qu'elle prend dans la méthode.

[1] LiONS se prononce « à l'anglaise » :[laɪəns].

Nous présentons ensuite le processus-même de la sélection des unités, et l'étape de synthèse qui la suit. Cette section se conclut par une description de l'évaluation qualitative réalisée par des auditeurs et par une analyse des avantages et des inconvénients de la méthode proposée.

10.1.1 Postulat et hypothèses

Notre postulat et nos hypothèses se fondent sur les points faibles relevés dans l'état de l'art, mais servent également un objectif supplémentaire : réduire le traitement du signal au strict minimum, c'est-à-dire au plus à un lissage en frontière d'unités.

Postulat 10.1.1 (Unité de base)**.** *La seule unité qui autorise un traitement du signal limité au point de concaténation est le diphone, étant donné que dans tous les cas, la concaténation est réalisée en partie stable du signal. L'apparente impossibilité de couvrir le diphone à l'aide d'un corpus de parole est simplement le reflet de la richesse réelle de la langue, que des unités non contextuelles comme le phone ou le demi-phone ne modélisent pas correctement.*

Hypothèse 10.1.1 (Critères de sélection)**.** *Un système de synthèse par sélection peut produire de la parole de haute qualité en n'utilisant* que *des informations linguistiques segmentales et suprasegmentales au niveau du coût cible, pour autant que le coût de concaténation soit* exclusivement *dirigé par des critères acoustiques.*

Hypothèse 10.1.2 (Pondération)**.** *Les critères linguistiques du coût cible dirigent une prise de décision prosodique. Ils sont donc de nature à être départagés par entropie. Les critères du coût de concaténation déterminent par contre une distance acoustique qu'il est nécessaire d'estimer en terme de qualité vocale. Ce domaine d'étude ne permet cependant pas encore de proposer une pondération automatique.*

10.1.1.1 Critères de sélection

Le seul support d'analyse de tout système de synthèse est le texte, mais le matériau qui est finalement manipulé est la parole. Un juste équilibre entre linguistique et acoustique doit donc diriger la sélection des unités de parole.

Coût cible. Le coût cible peut être dirigé à l'aide d'informations linguistiques segmentales et suprasegmentales, puisque ce sont ces informations qui permettent au module prosodique de générer les valeurs acoustiques (F0, durée) qui dirigent le coût cible des systèmes acoustiques. L'hypothèse est que les critères linguistiques, au contraire des critères acoustiques, dirigeront la courbe prosodique sans trop la contraindre. L'impératif, cependant, est d'utiliser suffisamment de critères suprasegmentaux *pertinents*, c'est-à-dire propices à décrire la courbe prosodique d'un énoncé.

Coût de concaténation. Etant donné que le processus de concaténation est totalement acoustique, le coût de concaténation ne peut reposer que sur des informations acoustiques. Toute information linguistique serait ici superflue, puisqu'elle ne permettrait en rien d'améliorer la perception de la *distance acoustique* entre deux unités.

10.1.1.2 Pondération

Etant donné que les critères du coût cible et du coût de concaténation sont de nature différente, il est inutile de vouloir les pondérer globalement.

Coût cible. Nous partons du constat que l'objectif des critères du coût cible est de choisir une *classe d'unités prosodiques*. Il s'agit donc d'une prise de décision, domaine dans lequel la théorie de l'information a montré l'intérêt de la notion d'*entropie*.

Coût de concaténation. L'état de l'art dans le domaine de la qualité vocale est encore assez limité : si les études s'accordent pour considérer que toutes les distances acoustiques ne sont pas perçues de manière similaire par l'oreille humaine, il est encore difficile, actuellement, de déterminer automatiquement les distances qui choquent et l'importance à accorder à chacune d'elles. Dans notre système, le coût de concaténation est dès lors estimé manuellement.

10.1.1.3 L'unité de base

Les systèmes de l'état de l'art n'ont pas retenu le diphone parce que sa couverture demande un corpus considérable. Or, le diphone présente l'avantage de contenir la phase de coarticulation, si difficile à modéliser.

Constats. Comme nous l'avons détaillé en Section 8.5.1, cette décision d'exclure le diphone repose simplement sur l'expression mathématique suivante :

$$m^2 \prod_{i=1}^{n} |F_i| \tag{10.1.1.1}$$

où m est le nombre de phonèmes d'une langue, n est le nombre de critères retenus et $|F_i|$ est le nombre de valeurs d'un critère particulier. En somme, cette formulation estime que tout diphone existe et que toutes les combinaisons des valeurs des différents critères du coût cible peuvent lui être attribuées. Or, ceci est linguistiquement faux :

1. Les contraintes articulatoires d'une langue donnée ne permettent pas la formation de toutes les paires de phonèmes de la langue. Prenons par exemple les semi-voyelles [w] et [ɥ] en français. Elles ne peuvent apparaître que devant voyelle. Elle ne peuvent donc commencer un diphone qui termine par une consonne ou par une autre semi-voyelle. Les diphones correspondants ainsi que toutes les combinaisons de critères associées peuvent être supprimés de la liste à couvrir.

2. En partant du principe que les critères retenus sont linguistiques, il est évident que certaines incompatibilités entre critères segmentaux et suprasegmentaux vont limiter les combinaisons possibles. Par exemple, un diphone |ba| pourra appartenir à une syllabe de type CV ou CVC, mais certainement pas V ou VC. De même, un diphone |ai| contient forcément la césure syllabique et n'appartient dès lors à aucun type de syllabe, à moins que l'on ne définisse une valeur particulière « césure ».

Une autre analyse corrobore ces faits linguistiques. Prenons un corpus constitué de phones et complet, dans le sens où toutes les combinaisons {phone+critères} y sont recensées. Admettons un système de synthèse utilisant ce corpus afin d'y rechercher des diphones accompagnés des mêmes critères. Une recherche {diphone+critères} qui ne trouverait aucun candidat *identique* mettrait simplement en évidence que les critères associés au phone ne sont pas suffisants, parce qu'ils ne proposent pas une couverture *complète* des combinaisons linguistiques réalisables. Typiquement, et en admettant que le diphone commence dans le phone courant et se termine dans le phone suivant, le critère qui manquera certainement, dans la définition d'un phone, est le phone suivant, puisque l'intégrer dans les critères reviendrait à estimer un corpus de diphones. Cette analyse démontre donc qu'un corpus de phones s'il couvre les critères choisis, n'est d'une taille raisonnable que s'il ne tient pas compte des *phénomènes de coarticulation*. Or, dans ce cas, un traitement du signal conséquent est nécessaire, ce que nous voulons éviter.

Réflexions. Ces constats nous incitent à considérer que le diphone n'est pas inconcevable en synthèse par sélection, parce que :

1. Le diphone est le seul à permettre une concaténation sur la partie stable du signal, au contraire du phone qui oblige à concaténer sur la phase de coarticulation et du demi-phone qui alterne entre partie stable et phase de coarticulation. Ce choix va donc dans le sens de la réduction du traitement du signal au strict nécessaire.
2. De nombreuses combinaisons sont linguistiquement irréalisables, ce qui diminue considérablement le champ des possibles. L'étendue de cette diminution dépend évidemment de la langue, des critères retenus et de l'incompatibilité de certaines de leurs valeurs.

L'utilisation de diphones implique cependant de prévoir l'absence totale d'un diphone donné, soit en revenant au niveau du demi-phone, soit en choisissant un diphone *proche*.

10.1.2 Le synthétiseur

LiONS a été conçu pour être intégré dans le système de synthèse *eLite*. Développé par les chercheurs du Centre de recherche (Beaufort & Ruelle 2006), eLite [2] signifie « *Enhanced, Linguistically-based Text-to-speech synthesis system* ». L'ensemble des données utilisées par eLite sont externalisées, de manière à rendre le système indépendant de la

[2] eLite se prononce [ilait].

langue. Les algorithmes des différents modules ne permettent cependant pas de traiter n'importe quelle langue : ils ont été prévus pour gérer les caractéristiques linguistiques des langues romanes et germaniques. La Figure 10.1 présente un schéma du module de traitement automatique de la langue d'eLite.

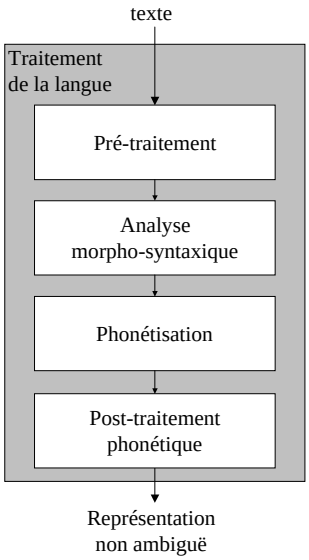


FIG. 10.1: eLite : module de traitement de la langue

Les principaux avantages de ce module sont :

- La qualité de son module de pré-traitement, capable de détecter un ensemble vaste d'unités complexes comme les URLs, les dates, les téléphones, les champs de données (comme les numéros de cartes bancaires). Ceci permet aux modules suivants d'adopter une approche *adaptée* au type de l'unité.
- La richesse de son analyse morpho-syntaxique, qui ne s'arrête pas à la simple détection d'une catégorie. Ce module adopte une approche statistico-linguistique dont il sera question dans la Partie III relative à la correction orthographique.
- La finesse des post-traitements phonétiques, point qui intéresse tout particulièrement le module de sélection :
  1. Vérification de la syllabation des mots hors-contexte. En cas de mauvaise syllabation, le mot est épelé.
  2. Gestion des phénomènes de liaison, qu'il s'agisse de l'insertion, de la suppression ou de la substitution d'un phonème (*les oiseaux*, *six mille*, *six amis*) ou de l'insertion de schwas épenthétiques afin de fluidifier le discours (*quelques personnes*, *quelques amis*).
  3. Syllabation contextuelle : la séquence phonétique de l'ensemble d'une phrase est divisée en syllabes, en tenant principalement compte de critères articulatoires.

10.1.3 Choix des critères

10.1.3.1 Critères du coût cible

Bien que l'unité de *sélection* soit le diphone, l'unité de *pré-sélection* est le phone dans son contexte phonique gauche et droit. Nous avons retenu 15 critères linguistiques, dont les valeurs sont listées dans la Table 10.1. La répartition de ces critères entre le phone et ses contextes est détaillée par la Table 10.2 : en tout, 36 critères ($10 + 2 * 13$) sont utilisés pour la pré-sélection d'une unité.

Notre objectif, en retenant tant de critères, était de pouvoir confirmer ou infirmer de manière définitive l'Hypothèse 10.1.1. Moins de critères, en cas d'échec, nous auraient certainement conduits à la conclusion facile qui veut que « la méthode est prometteuse, mais les bons critères n'ont probablement pas été retenus ».

Nous proposons tout d'abord une vue d'ensemble des critères sélectionnés. Nous expliquons ensuite l'intérêt linguistique des critères suprasegmentaux.

Vue d'ensemble. On constate (cf. Table 10.2) que seuls les contextes reçoivent des critères segmentaux : ce sont les caractéristiques articulatoires, qui permettent de définir les contextes sans recourir au nom du phonème. Ceci permet de donner une certaine latitude à la pré-sélection, qui peut de la sorte choisir un phone cible entouré de contextes coarticulatoires proches, mais différents des phones recherchés.

Tous les autres critères sont suprasegmentaux et concernent, dans un ordre croissant, la syllabe, le mot, le groupe et la phrase.

Groupe et position dans le groupe. Le groupe que nous employons n'est pas le syntagme de l'analyse syntaxique. Comme nous aurons l'occasion de la détailler (cf. Partie III), notre analyse syntaxique se limite à choisir la meilleure suite de catégories pour une phrase donnée, sans construire la structure arborescente syntaxique de la phrase. Nos groupes sont construits dans un second temps, selon une approche très semblable à celle de l'algorithme *chinks 'n chunks*.

Le principe *chinks 'n chunks* (Liberman & Church 1991) est de séparer les catégories entre la classe des *chinks* ou *mots-fonctions* (déterminant, préposition...) et celle des *chunks* ou *mots-contenus* (nom, verbe...). Un groupe, pour être valide, doit ensuite correspondre à l'expression suivante : (*chinks** *chunks**). En somme, dès qu'un *chunk* est suivi par un *chink*, une frontière de groupe sépare les deux mots. Notre approche est un simple raffinement de cet algorithme. Nous la détaillons en Annexe 1. Les groupes constitués par notre algorithme s'appellent *head1* (tête nominale), *head2* (tête verbale), *alone* (catégorie seule) et *punct* (ponctuation).

Nous employons les *chinks 'n chunks* pour diriger la courbe prosodique de l'énoncé. Les *chinks 'n chunks* ont déjà été employés en modélisation prosodique. Par exemple, Malfrère *et al.* (1998) y font correspondre leurs groupes intonatifs, et attribuent en français un

	Critère	#	**Valeurs**
1.	Voisement	2	non-voisé, voisé
2.	Type Articulation	3	consonne, voyelle, semi-voyelle
3.	Lieu Articulation	14	bilabial, labiodental, ..., latéral
4.	Mode Articulation	5	oral, nasal, plosif, constrictif, liquide
5.	Labialité	3	indéfini, étiré, arrondi
6.	Accent tonique	3	0 (non accentué), 2 (secondaire), 1 (primaire)
7.	Struct. syllabique	5	V, CV, VC, CVC, silence
8.	Pos./syllabe	3	onset, nucleus, coda
9.	Long. Syllabe	7	de 1 à 7
10.	Pos. Syllabe/mot	3	début, dedans, fin
11.	Long. Mot	3	court, moyen, long
12.	Pos. Mot/groupe	3	début, dedans, fin
13.	Groupe	4	head1, head2, alone, punct
14.	Pos. Mot/phrase	3	début, dedans, fin
15.	Long. Phrase	3	court, moyen, long

TAB. 10.1: Critères du coût cible et valeurs en français

	Cat.	**Critères**	**gauche**	**cible**	**droit**
1.	seg.	Voisement	v		v
2.		Type Articulation	v		v
3.		Lieu Articulation	v		v
4.		Mode Articulation	v		v
5.		Labialité	v		v
6.	sup.	Accent tonique	v	v	v
7.		Struct. syllabique	v	v	v
8.		Pos./syllabe		v	
9.		Long. Syllabe	v	v	v
10.		Pos. Syllabe/mot	v	v	v
11.		Long. Mot	v	v	v
12.		Pos. Mot/groupe	v	v	v
13.		Groupe	v	v	v
14.		Pos. Mot/phrase	v	v	v
15.		Long. Phrase		v	

seg.=*segmental*, sup.=*suprasegmental*

TAB. 10.2: Répartition des critères du coût cible

accent de fin de groupe à la dernière syllabe de chaque groupe intonatif.

Dans notre cas, nous qualifions les *chinks 'n chunks* de *groupes rythmiques*.

Définition 10.1.1. *Un* groupe rythmique *est un tout cohérent. Il constitue un groupe de souffle, susceptible d'être suivi d'une pause et de porter un accent. Il peut lui-même être constitué de groupes rythmiques, selon le rythme d'élocution adopté.*

La nuance est importante, puisque nous n'obligeons en aucun cas la sélection à trouver un accent défini. Nous signalons simplement au module de pré-sélection que nous désirons une unité située à un emplacement donné d'un groupe rythmique.

La définition des groupes rythmiques a également été l'occasion de redéfinir la syllabation d'une phrase. Le processus de syllabation construit toujours les syllabes d'une phrase en respectant les contraintes articulatoires de la langue, mais ne peut jamais créer une syllabe qui contienne une frontière de groupe. Prenons un exemple :

Il est multilingue en tout cas.

Une syllabation standard donnera :

il	*est*	*multilingue*	‖	*en*	*tout*	*cas*
i	l e	m y l t i l ɛ̃		**g** ɑ̃	t u	k a

Notre contrainte supplémentaire oblige le système à produire :

il	*est*	*multilingue*	‖	*en*	*tout*	*cas*
i	l e	m y l t i l ɛ̃ **g**		ɑ̃	t u	k a

Dans notre exemple, le [g] devient implosif au lieu d'être explosif. L'objectif est bien sûr qu'à l'écoute, cette modification améliore la compréhension en permettant de mieux percevoir les groupes, sans introduire de pause dans le signal.

Les groupes rythmiques servent donc 2 objectifs :

1. Ajouter une information prosodique nécessaire.
2. Adapter la syllabation dans le contexte de la phrase.

Accent tonique. Une description de l'accent tonique a été donnée en Section 8.3.2. Notre hypothèse est que l'accent tonique *au niveau du mot* est un critère primordial, quelle que soit la langue.

Une remarque est à faire concernant le français. Section 8.3.2, nous avons signalé que les phonéticiens s'accordent pour dire que « l'accent de mot disparaît au profit de l'accent de groupe ». De ce fait, les modèles prosodiques du français utilisés en synthèse, comme par exemple (Malfrère *et al.* 1998), placent généralement un accent final sur la dernière syllabe accentuée du groupe, toutes les autres syllabes étant non accentuées (cf. Figure 10.2).

le	temps	de	la	parenthèse			était			arrivé	
l ə	t ɑ̃	d ə	l a	p a	ʁ ɑ̃	t ɛ z	e	t ɛ	t a	ʁ i	v e
NA	NA	NA	NA	NA	NA	**AF**	NA	NA	NA	NA	**AF**

FIG. 10.2: Accents toniques d'un modèle prosodique. *NA*=« non accentué », *AF*=« accent final »

Nous proposons un autre point de vue, basé sur l'analyse de spectrogrammes de signaux de parole. Il nous semble que les zones du signal correspondant à un accent de mot présentent généralement une énergie et une amplitude supérieures aux zones du signal correspondant à des syllabes non accentuées, que l'accent de mot corresponde ou non à l'accent de groupe. Sur la base de cette analyse, nous posons le postulat suivant :

Postulat 10.1.2 (Accent de mot)**.** *En français, l'accent de mot est conservé, mais s'atténue par rapport à l'accent de groupe. Deux* degrés *d'accentuation existent donc en français : un degré* majeur *au niveau du groupe et un degré* mineur *au niveau du mot.*

Du fait de ce postulat et quelle que soit la langue, nous gardons tous les accents secondaires et primaires de la phrase. Dans notre modèle, l'*intensité* d'un accent dépend donc de deux critères : la valeur de l'accent et la position du mot dans le groupe. Cette intensité n'est cependant pas fixée, puisque nous ne convertissons pas nos critères linguistiques en valeurs acoustiques (cf. Figure 10.3).

le	temps	de	la	parenthèse			était			arrivé	
l ə	t ɑ̃	d ə	l a	p a	ʁ ɑ̃	t ɛ z	e	t ɛ	t a	ʁ i	v e
0	**1**	0	0	**2**	0	**1**	**2**	**1**	**2**	0	**1**
beg	in	in	in	**end**	**end**	**end**	**beg**	**beg**	**end**	**end**	**end**

FIG. 10.3: Niveaux d'accentuation. Le premier niveau est l'accent tonique : non accentué (0), secondaire (2), primaire (1). Le second niveau est la position dans le groupe rythmique : début (beg), dedans (in), fin (end)

Conserver tous les accents présente un autre avantage. En effet, la durée des voyelles en français dépend entre autres de l'accent. Les syllabes non accentuées sont toujours brèves, tandis que les syllabes accentuées peuvent être brèves ou longues. L'accent de mot permet donc de distinguer la durée d'une syllabe atone de celle d'une syllabe portant un accent (primaire ou secondaire) de degré mineur.

Structure syllabique. La Table 10.3 détaille l'influence de la fermeture ou de l'ouverture de la syllabe sur le timbre et la durée de la voyelle (Pierret 1994).

En ce qui concerne le timbre, on notera que la transcription phonologique nous renseigne déjà, comme dans le cas du « o » fermé dans *peau* ([p o]) et ouvert dans *bol* ([b ɔ

l]). Cependant, en fonction de la structure de la syllabe, un son fermé peut avoir tendance à s'ouvrir, et un son ouvert peut avoir tendance à se fermer.

Il est évident que cette influence de la structure syllabique est une *tendance* et non une généralité. Cette information paraît cependant pertinente, pour autant qu'on évite, contrairement aux modèles prosodiques, de la convertir automatiquement en valeurs acoustiques *constantes*.

Position dans la syllabe. Nous avons détaillé en Section 8.3.2 ce qu'est une syllabe, les parties qu'elle peut comporter, les structures classiques qui en découlent et l'influence de la syllabe sur les traits suprasegmentaux. Cette influence montre l'intérêt de posséder l'information dans la recherche d'une cible, tant au niveau de la qualité du timbre que de l'énergie.

Longueur de la phrase. Nous faisons l'hypothèse que la vitesse d'élocution est influencée par la longueur de la phrase, du fait de la nécessité de gérer son souffle. Une phrase longue devrait inciter le locuteur à accélérer le rythme. Nous avons empiriquement déterminé trois longueurs en nombre de syllabes :

- Phrase courte : maximum 5 syllabes.
- Phrase moyenne : entre 5 et 15 syllabes.
- Phrase longue : plus de 15 syllabes.

Autres critères. D'autres critères, comme la longueur de la syllabe, ont été ajoutés afin d'affiner la recherche, mais sans intuition linguistique de l'apport réel qu'ils peuvent représenter.

10.1.3.2 Critères du coût de concaténation

Les critères que nous avons retenus sont trois critères assez classiques, la F0, le spectre et l'énergie, que nous complétons par la durée du demi-phone. Nous rappelons que pendant l'étiquetage du corpus, la F0, le spectre et l'énergie ont été prélevés dans chaque phone au point de frontière diphonique. Un diphone présente donc ces informations à chaque frontière.

F0. Nous évaluons la distance de F0 en mel, une échelle de mesure proposée par Stevens *et al.* (1937) dans laquelle une différence de n Hz entre deux fréquences est ressentie de manière identique par l'auditeur quelle que soit la zone de fréquences où elle se produit. La conversion d'une F0 f en mel est réalisée comme suit :

$$2595 \log \left(1 + \frac{f}{700}\right) \tag{10.1.3.1}$$

	Syllabe fermée			**Syllabe ouverte**
	voyelles nasales	[v], [z], [ʒ], [ʁ]	autres	
Timbre	ouvert			fermé
Durée	long		bref	

TAB. 10.3: Influence de la structure syllabique sur le timbre et la durée

La distance entre deux F0 f_1 et f_2 se calcule comme la différence absolue de leurs valeurs en mel :

$$D_{F0} = 2595 * \text{abs}\left(\log\left(1 + \left(\frac{f_1}{700}\right)\right) - \log\left(1 + \left(\frac{f_2}{700}\right)\right)\right) \tag{10.1.3.2}$$

Spectre. Il est relativement difficile d'évaluer dans quelle mesure une différence spectrale entre deux unités concaténées va gêner l'oreille humaine. Les résultats de plusieurs méthodes d'évaluation ont fait l'objet de tests à l'audition (Klabbers & Veldhuis 1998, Donovan 2001), dont il ressort que la distance perceptuelle de Kullback-Leibler est l'une des mesures les plus fiables. Initialement, Kullback-Leibler est une mesure statistique de dissimilarité entre deux distributions de probabilités P et Q (Kullback & Leibler 1951) :

$$\text{KL}(P \parallel Q) = \int p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x \tag{10.1.3.3}$$

où p et q dénotent les densités de P et Q. Or, $\text{KL}(P \parallel Q)$ ne correspond pas à $\text{KL}(Q \parallel P)$. Il s'agit d'une *divergence* et non d'une distance. Pour obtenir une distance, on somme les deux divergences et on divise par 2 :

$$D_{\text{KL}}(P, Q) = \frac{\text{KL}(P \parallel Q) + \text{KL}(Q \parallel P)}{2} \tag{10.1.3.4}$$

Dans le cas de l'évaluation de la distance entre deux spectres v_1 et v_2, nous disposons de n coefficients par spectre (256 dans notre cas). L'intégration peut dès lors prendre la forme d'une somme :

$$D_{Spec} = \frac{FE}{2n} \sum_{i=1}^{n} v_1[i] - v_2[i] \cdot \log\left(\frac{v_1[i]}{v_2[i]}\right) \tag{10.1.3.5}$$

où FE est la fréquence d'échantillonnage, 16KHz dans notre cas.

Cette distance est tout spécialement adaptée à l'estimation de la différence entre deux spectres prélevés sur deux réalisations acoustiques d'un *même* phonème. Ceci a donc retenu notre attention, étant donné que notre unité de concaténation, le diphone, se concatène sur des demi-phones appartenant à un même phonème.

Energie. La distance entre deux énergies E_1 et E_2 correspond à leur différence absolue :

$$D_E = \text{abs}(E_1 - E_2) \tag{10.1.3.6}$$

Durée du demi-phone. Ce critère est ajouté parce que notre unité de concaténation est le diphone. Lors de la concaténation de deux diphones, la jonction est réalisée entre deux demi-phones d'un même phonème. Nous sommes en partie stable, ce qui facilite la concaténation. Par contre, les deux demi-phones peuvent être de *durée* différente. Or, la concaténation de deux demi-phones qui diffèrent fortement en durée donne une impression de changement brutal de rythme, que l'oreille humaine a tendance à confondre avec une erreur classique de concaténation. Il est dès lors nécessaire de limiter les différences de durée entre demi-phones afin d'obtenir un résultat naturel. Pour deux diphones |ba| et |ac| à concaténer sur /a/ (/a| · |a/), la différence de durée des demi-phones vaut :

$$D_{Dur} = \text{abs}(\texttt{Right}(|\text{ba}|) - \texttt{Left}(|\text{ac}|)) \tag{10.1.3.7}$$

où `Right` note la durée du demi-phone droit, et `Left`, la durée du demi-phone gauche.

10.1.4 Corpus et entraînement

Le corpus que nous avons utilisé a été prêté au Centre de recherche par la société Acapela [3], anciennement Babel Technologies, à des fins de recherche dans le domaine de la synthèse par sélection.

Ce corpus est constitué de 950 phrases extraites du journal Le Soir [4] et correspond à environ 1 heure de parole [5]. Il a été enregistré à une fréquence d'échantillonnage de 16 KHz, en 16 bits et en mono, dans un environnement calme mais non anéchoïque. La locutrice est francophone et de nationalité suisse.

Lorsque nous avons reçu ce corpus, nous disposions pour chaque phrase de trois fichiers, le premier contenant la parole (format *wav*), le second, le texte et le troisième, la séquence phonétique. Nous parlons respectivement du fichier son, du fichier texte et du fichier phonétique.

L'entraînement sur le corpus comprend trois étapes :

1. L'alignement phonétique et l'analyse acoustique du corpus
2. La création de la table d'étiquetage phonétique
3. L'entraînement de la pondération

[3] www.acapela-group.com.
[4] Journal belge francophone : www.lesoir.be.
[5] Le corpus représente en fait 1 heure 15 de parole, mais 15 minutes peuvent être supprimées si l'on enlève les longs silences de début et de fin de phrase.

10.1.4.1 Alignement phonétique et analyse acoustique

Alignement phonétique. L'alignement entre la séquence phonétique et la parole a été obtenu au moyen du système de reconnaissance de la parole *STRUT* (Boîte *et al.* 1997), dont les principaux algorithmes sont présentés dans (Deroo *et al.* 1996, 1997, Dupont & Ris 2003, Dupont *et al.* 2005). Il s'agit d'un système de reconnaissance hybride HMM[6]/ANN[7], dont la particularité est que la première phase de la reconnaissance, le classement des trames acoustiques, est réalisée à l'aide d'un réseau de neurones. Le réseau de neurones charge un modèle construit au cours d'un entraînement à partir d'un corpus phonétiquement aligné. En l'occurrence, le modèle a été entraîné sur 100 heures de textes lus provenant du corpus Bref (Lamel *et al.* 1991).

Afin d'obtenir un alignement de qualité, nous avons réalisé un premier alignement automatique à l'aide du reconnaisseur. Trois chercheurs ont ensuite corrigé manuellement l'alignement obtenu. La visualisation et la correction de l'alignement a été réalisée à l'aide du logiciel Praat (Boersma 2003), outil en distribution libre sur Internet [8] destiné à l'édition de signaux de parole pour les phonéticiens. Ce logiciel permet de représenter plusieurs niveaux d'étiquetage décrits par l'utilisateur [9]. En ce qui nous concerne, un étiquetage en phonèmes a été suffisant (cf. Figure 10.4). Cet outil, très pratique, a certainement contribué à réduire le temps nécessaire à la correction. Néanmoins, il a fallu trois semaines aux trois chercheurs afin de corriger la totalité du corpus...

A titre indicatif, le taux de reconnaissance, lorsque l'on compare l'alignement automatique à l'alignement corrigé, est de 68,63%. Nous avons établi ce taux en considérant comme équivalentes deux frontières posées dans un intervalle de maximum 20 ms. L'écart entre l'alignement automatique et manuel montre qu'une phase de correction manuelle est primordiale.

Cependant, la totalité de l'alignement du corpus ne doit pas forcément être corrigée manuellement. Le Table 10.4 montre l'évolution du taux de reconnaissance en fonction du nombre de phrases du corpus sur lesquelles le reconnaisseur a été surentraîné. Un critère très important pour notre système, étant donné qu'il utilise des diphones, est la position du milieu de phonème, lieu ou la segmentation diphonique est réalisée. Lorsque l'on estime la corrélation entre alignement automatique et manuel sur la position du milieu de phonème, on constate que l'alignement automatique devient acceptable dès un surentraînement sur 100 phrases du corpus. Le temps d'alignement manuel peut donc être fortement réduit.

L'alignement final de chaque phrase est conservé dans un fichier ali, dans lequel chaque ligne correspond à un phone et recense le nom du phone, ainsi que le début et la fin du phone en nombre d'échantillons (cf. Section 8.2).

[6] HMM : *Hidden Markov Model*, « modèle de Markov caché ».
[7] ANN : *Artificial Neural Network*, « réseau de neurones artificiels ».
[8] `www.fon.hum.uva.nl/praat`.
[9] Les phonéticiens emploient classiquement un étiquetage en phonèmes, en syllabes, en tons et en mots.

Surentraînement (Nb. phrases)	**Sur la frontière de phonème (%)**	**Sur le milieu de de phonème (%)**
0	68,63	85,56
100	86,67	92,4
300	87,72	92,33
500	92,11	96,28
600	92,33	96,54
totalité	93,25	97,13

TAB. 10.4: Evolution du taux de reconnaissance en fonction du surentraînement réalisé

Analyse acoustique. Le processus d'alignement automatique d'un fichier son se termine par le prélèvement d'informations acoustiques sur le signal toutes les 8 ms. Ces informations sont la F0, les coefficients LPC et l'énergie. Trois nouveaux fichiers sont donc créés pour chaque phrase : le fichier f0, le fichier lpc et le fichier nrg.

10.1.4.2 Création de l'étiquetage

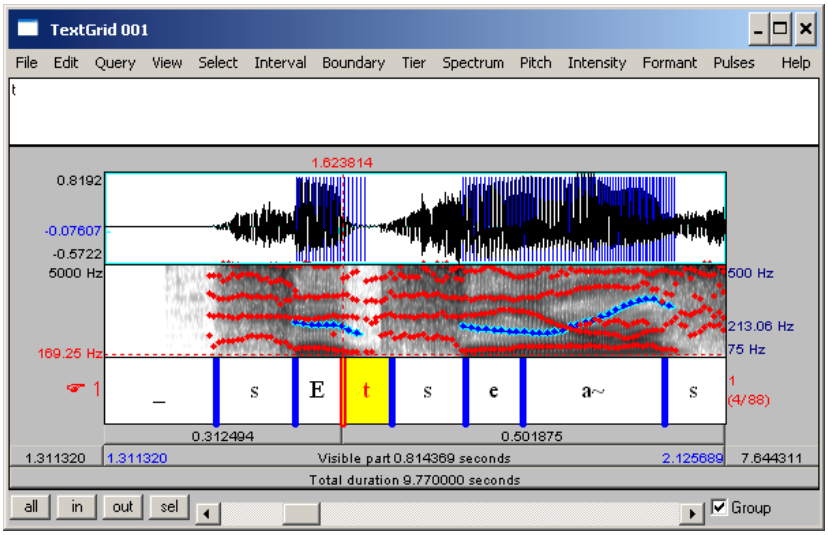
Dans la première version de LiONS et bien que la sélection choisisse des diphones, la table d'étiquetage générée est de type phonétique et non diphonique.

Pour chaque phrase, le processus de construction de la table consulte quatre fichiers : texte, ali, f0 et nrg. Pour une phrase i, les étapes du processus sont :

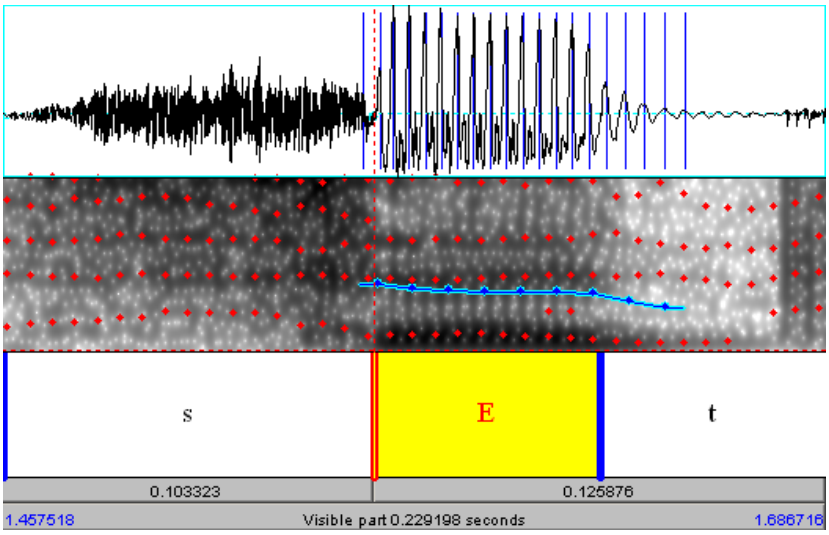
1. Les modules d'eLite, depuis le pré-traitement jusqu'au post-traitement phonétique, analysent le fichier texte i afin de produire les critères linguistiques du coût cible
2. Pour chaque phonème de la phrase, on ajoute une ligne dans la table d'étiquetage recensant :
   (a) Le nom du phone
   (b) Les valeurs des critères linguistiques générées par l'analyse d'eLite.
   (c) Le numéro du fichier, qui correspond au numéro de la phrase en cours de traitement.
   (d) Le début et la fin du phone (en nombre d'échantillons) dans le fichier son, lus dans le fichier ali.
   (e) La F0 moyenne et l'énergie moyenne du phone, générées à partir des valeurs sauvées pour le phone dans les fichiers f0 et nrg.

La table d'étiquetage finale ressemble au schéma présenté en Annexe 2.

Cohérence. Nous devons signaler que l'étape (1) du processus d'étiquetage a dû être adaptée. Le corpus de parole n'a pas été construit par eLite. De ce fait, des divergences



(1)



(2)

FIG. 10.4: Praat : vue d'ensemble(1) et zoom sur l'alignement(2)

peuvent se produire entre la séquence phonétique générée par eLite au cours de l'analyse, et la séquence phonétique du fichier ali. Les divergences sont de trois types :

1. Divergence de timbre. Par exemple, une opposition /e/–/ɛ/.
2. Divergence de liaison ou de schwa épenthétique. Par exemple, une liaison facultative peut avoir été réalisée par eLite ou par la locutrice, mais pas par les deux en même temps.
3. Divergence régionale. La locutrice est suisse et emploie certains régionalismes tels que « octante » pour « quatre-vingts ».

Dans tous les cas, le fichier ali a raison et eLite *doit* s'adapter, afin que la table d'étiquetage soit cohérente avec le corpus oral. En ce qui concerne les divergences (1) et (2), une simple comparaison des séquences phonétiques est réalisée pour chaque mot. S'il y a divergence, les phonèmes concernés sont remplacés par ceux du fichier ali. En ce qui concerne les divergences (3), des bases de données suisses ont été générées et ont été chargées au lancement d'eLite en lieu et place des bases de données du français standard.

10.1.4.3 Entraînement de la pondération

A partir des informations mémorisées dans la table d'étiquetage et des fichiers acoustiques qui ont été générés au cours de l'alignement, nous avons mis en place la méthode de pondération automatique des critères du coût cible et choisi manuellement des pondérateurs pour les critères du coût de concaténation. Au vu de l'importance de cette étape, nous lui avons entièrement consacré la Section suivante.

10.1.5 Pondération des critères

10.1.5.1 Les critères linguistiques

Nous partons de l'hypothèse suivante :

Hypothèse 10.1.3 (Pondération par phonème)**.** *Du fait de leurs différences articulatoires, les phonèmes se comportent différemment les uns des autres dans un même contexte d'élocution. De ce fait, une seule pondération des caractéristiques linguistiques ne serait pas pertinente. Il est préférable de pondérer les caractéristiques pour chaque phonème indépendamment.*

Nous proposons donc une méthode qui détermine les poids des critères du coût cible *par phonème*. Cette méthode, selon notre Hypothèse 10.1.2, repose sur la notion d'entropie.

Principe. Nous considérons que les réalisations acoustiques d'un phonème dans le corpus peuvent être réunies en n classes de réalisations acoustiquement similaires. Une classe C_i a

une probabilité d'émission *a priori*, définie comme le rapport entre le nombre de réalisations de la classe et la somme du nombre de réalisations des n classes du phonème :

$$P(C_i) = \frac{|C_i|}{\sum_{j=1}^{n} |C_j|} \tag{10.1.5.1}$$

Dans l'absolu, il y a donc une part d'aléatoire dans l'émission de ces classes, aléatoire auquel la théorie de l'information donne le nom d'*entropie* (Shannon 1948). L'entropie est au départ une notion physique qui définit l'état d'agitation d'un système. En théorie de l'information, elle désigne le degré d'information d'un système : plus l'entropie du système est élevée, et moins ce système comporte d'information (il est désordonné). L'entropie d'un système S se calcule en nombre de bits nécessaires pour coder l'information :

$$H(S) = -\sum_{i=1}^{n} P(C_i) \cdot \log_2 P(C_i) \tag{10.1.5.2}$$

Puisque les réalisations acoustiques des classes du système sont accompagnées de m critères distincts, on peut espérer que chacun de ces critères apporte une information pertinente qui diminue l'entropie du système (Quinlan 1979). Le gain d'information apporté par un critère F_j dépend des valeurs v_i qu'il peut prendre et de la répartition de ces valeurs entre les différentes classes du système. Pour connaître le gain d'information apporté par un critère F_j, il faut donc commencer par calculer son entropie :

$$H(F_j) = \sum_{i=1}^{m} P(v_i) \cdot H(S|v_i) \tag{10.1.5.3}$$

Le gain d'information de F_j est ensuite simplement :

$$G(F_j) = H(S) - H(F_j) \tag{10.1.5.4}$$

Plus $G(F_j)$ est élevé, et plus le critère F_j réduit l'aléatoire dans la prise de décision.

Le gain d'information est une mesure pertinente. Il a cependant tendance à favoriser les critères qui possèdent de nombreuses valeurs *différentes*. On parle de *division de l'information*, que l'on mesure comme suit :

$$DI(F_j) = -\sum_{i=1}^{m} P(v_i) \cdot \log_2 P(v_i) \tag{10.1.5.5}$$

L'influence de la division de l'information sur le gain d'information peut dès lors être neutralisée en calculant le rapport de gain (Quinlan 1986) :

$$GR(F_j) = \frac{G(F_j)}{DI(F_j)} \tag{10.1.5.6}$$

Plus $GR(F_j)$ est élevé, et plus le critère F_j réduit l'aléatoire dans la prise de décision.

Traditionnellement, le gain d'information et son dérivé le rapport de gain sont utilisés afin de répartir les critères F_j d'un système S dans un arbre de décision (Breiman *et al.* 1984, Quinlan 1993, van den Bosch 1997), en plaçant les critères dans l'ordre décroissant de leur importance. Les feuilles de l'arbre sont les classes C_i du système, et chaque nœud intermédiaire de l'arbre retient également la classe la plus probable à son niveau. Ceci permet de parcourir l'arbre avec un vecteur de critères dont les valeurs ne sont pas toutes dans l'arbre : à un nœud de l'arbre correspondant au critère F_j, si la valeur de F_j dans le vecteur ne correspond à aucune des valeurs de l'arbre, le nœud retourne la classe par défaut qu'il a mémorisée.

Le parcours d'un arbre est très rapide, et le système de décision par défaut est certainement pertinent dans de nombreux domaines, comme celui de la phonétisation automatique. Par contre, ce mécanisme de recherche ne permet pas d'employer les poids qui ont permis la répartition des critères dans l'arbre. Pour cette raison, nous avons préféré éviter la construction de l'arbre, et simplement utiliser le rapport de gain de chaque critère F_j afin de déterminer son poids $W(F_j)$ dans le coût cible :

$$W(F_j) = 2\ \log(1 + 10\, GR(F_j)) \tag{10.1.5.7}$$

Notons que la fonction logarithmique qui intervient dans le calcul du poids permet de lisser les valeurs obtenues.

Algorithme. Pour chaque phonème,

1. Récolter l'ensemble des réalisations acoustiques du phonème.
2. Rassembler les réalisations acoustiques en n classes selon un critère de similarité acoustique, de manière à créer un système S.
3. Calculer $H(S)$.
4. Pour chaque critère F_j du coût cible :
   (a) Calculer $GR(F_j)$.
   (b) Attribuer à F_j le poids $W(F_j)$.

Construction des classes acoustiques. Le point (2) de l'algorithme décrit ci-dessus demande une méthode permettant de créer des classes de réalisations acoustiques selon un critère de similarité. Afin d'initialiser le processus, notre premier critère de similarité est la durée des réalisations : nous créons k classes, maximum 7, dans lesquelles les réalisations sont réparties en fonction du rapport que leur durée d entretient avec la moyenne $\overline{m}$ et l'écart type σ des durées de l'ensemble des réalisations :

1. $d \leq \overline{m} - 2\sigma$
2. $\overline{m} - 2\sigma < d \leq \overline{m} - \sigma$
3. $\overline{m} - \sigma < d \leq \overline{m} - \frac{\sigma}{2}$

4. $\overline{m} - \frac{\sigma}{2} < d \leq \overline{m} + \frac{\sigma}{2}$
5. $\overline{m} + \frac{\sigma}{2} < d \leq \overline{m} + \sigma$
6. $\overline{m} + \sigma < d \leq \overline{m} + 2\sigma$
7. $\overline{m} + 2\sigma < d$

Cette partition est ensuite raffinée à l'aide de l'algorithme K-Means (McQueen 1967, Forgy 1965). Le principe de cet algorithme est qu'il estime de manière itérative la pertinence d'une partition en n classes, n variant entre deux bornes fixes. La meilleure partition est celle qui minimise la variance inter-classe tout en maximisant la variance intra-classes. En ce qui concerne la variance intra-classe, elle est obtenue sur la base de la distance entre les éléments de la classe, comparés deux à deux. En ce qui concerne la variance inter-classes, elle est obtenue sur la base de la distance entre les centroïdes [10] des classes, comparés deux à deux.

Dans notre cas, la distance entre éléments est estimée à l'aide d'un indice de similarité acoustique. En l'occurrence, nous avons choisi la distance perceptuelle de Kullback-Leibler, que nous avons déjà utilisée pour calculer la distance entre deux spectres dans le coût de concaténation (cf. Section 10.1.3.2). Comme nous l'avons mentionné, cette distance est tout spécialement adaptée à l'estimation de la différence entre deux réalisations acoustiques d'un même phonème.

Nous imposons au K-Means que le nombre de classes se situe entre 5 et 120.

10.1.5.2 Les critères acoustiques

Les poids attribués aux différents critères acoustiques du coût de concaténation (cf. Table 10.5, a) ont été choisis expérimentalement, sur un petit corpus de phrases. Or, dans cette première version de LiONS, les valeurs des critères acoustiques n'ont pas été normalisées (cf. Table 10.5, b). Les poids ne sont dès lors pas représentatifs de la part attribuée à chaque critère dans le coût. Ainsi, la F0 est fortement favorisée dans le cas d'un son voisé (cf. Table 10.5, c), alors que son poids semble la désavantager par rapport au spectre. Notons que le spectre est le critère le plus important dans le cas d'un son non voisé, étant donné que la F0 est nulle.

10.1.5.3 Pondération globale du double coût

Cette pondération globale « coût cible – coût de concaténation » est également manuelle. A partir du même petit corpus de phrases, nous avons expérimentalement décidé de multiplier le coût cible par 5 :

$$U = \arg\min \sum_{j=1}^{l} 5 \times \texttt{CI}(u_j|c_j) + 1 \times \texttt{CO}(u_{j-1}, u_j) \tag{10.1.5.8}$$

[10] Le centroïde peut être soit un élément fictif, calculé comme étant la représentation moyenne des éléments de la classe, soit l'élément de la classe considéré comme le plus proche de l'élément fictif précédent.

Cependant, l'absence de normalisation du coût de concaténation ne permet pas d'affirmer que le coût cible est effectivement avantagé dans le système.

10.1.6 Processus de sélection et synthèse

10.1.6.1 Processus de sélection

Dans l'ensemble, le processus est classique : une phase de pré-sélection précède la phase de sélection proprement dite.

Pré-sélection. Nous avons signalé que si l'unité de concaténation est le diphone, l'unité de pré-sélection est le phone. Pour chaque phonème de la phrase, nous créons donc une cible qui est formée du nom du phonème et des 36 critères linguistiques que nous avons détaillés précédemment.

Dans cette première version de LiONS, la pré-sélection n'est pas optimisée : pour une cible donnée, la recherche est réalisée de manière séquentielle dans la table d'étiquetage. Le résultat est l'ensemble des phones du corpus qui portent au moins le nom du phonème cible. Lorsque la totalité des cibles ont été recherchées, un premier élagage est réalisé : étant donné une liste de candidats pour une cible donnée, nous ne conservons que les candidats dont le phone de contexte droit permet la création du diphone désiré, pour autant qu'au moins un diphone désiré soit réalisable. Dans le cas contraire, toutes les possibilités de la liste sont conservées (cf. Figure 10.5).

Les diphones sont ensuite pondérés de manière à pouvoir réaliser un second élagage qui ne conserve au plus que les 40 meilleurs candidats. Le coût d'un diphone candidat est simplement la somme, divisée par deux, des coûts des deux phones candidats dont il est extrait :

$$\mathtt{CI}_{di}(u_{i^k,j^l}|c_{i,j}) = \frac{\mathtt{CI}_{pho}(u_{i^k}|c_i) + \mathtt{CI}_{pho}(u_{j^l}|c_j)}{2} \tag{10.1.6.1}$$

Ceci revient à attribuer aux critères du diphone une pondération égale à la moyenne des pondérations des critères des deux phones qui le constituent.

Sélection. L'ensemble des diphones candidats pour l'ensemble des diphones cibles constituent un treillis de solutions. Ce treillis est classiquement résolu par programmation dynamique, en recherchant la séquence de diphones qui minimise le double coût pondéré « cible – concaténation ».

La programmation dynamique (Bellman & Dreyfus 1962) est une approche qui permet de rapidement trouver le meilleur chemin entre un nœud initial I et un nœud final F dans un treillis de possibilités, en évitant de calculer le coût de tous les chemins du treillis entre I et F. L'idée est de considérer la solution globale comme la suite des meilleures hypothèses locales. Dans ce modèle, le coût d'un nœud i du treillis depuis le nœud I vaut :

$$\mathtt{C}(i, I) = w(i) + \min_{h=1}^{n}\{d(i, h) + \mathtt{C}(h, I)\} \tag{10.1.6.2}$$

Critères	**(a) Poids**	**(b) médiane**		**(c) médiane * poids**	
		[i]	[e]	[i]	[e]
F0	200	68.6009	66.7055	13 720.18	13 341.1
Spectre	200 000	0.0036	0.0027	720.	540.
Energie	2	9.4525	9.3140	85.0725	18.628
Durée	0.125	202.	168.	25.25	21.

TAB. 10.5: Coût de concaténation : poids et valeurs des critères pour les phonèmes *i* et *e*. LA F0 est exprimée en mel et la durée, en échantillons

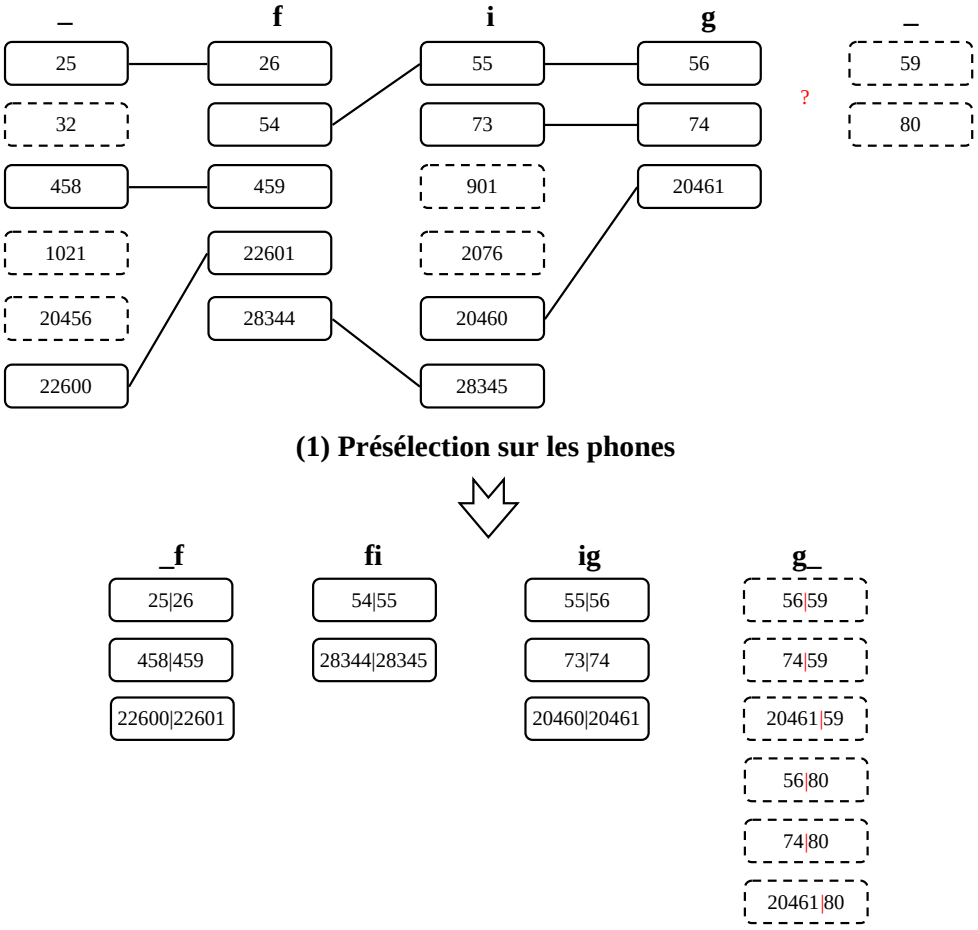


FIG. 10.5: Pré-sélection. On cherche la séquence /_ f i g _/. Un premier élagage est réalisé en fonction des diphones existants

où $w(i)$ note le poids du nœud i, h est un prédécesseur de i, n est le nombre de prédécesseurs de i, et $d(i,h)$ représente le coût de transition entre i et h. La solution globale correspond à $\mathtt{C}(F,I)$, et peut être construite pour autant que l'on ait conservé, dans chaque nœud, le meilleur de ses prédécesseurs.

Dans notre sélection,

- Un nœud i correspond à un diphone candidat $u_{i[k],j[l]}$ pour un diphone cible $c_{i,j}$,
- $w(i)$ correspond au coût cible,
- $d(i,h)$ est le coût de concaténation.

Format de sortie. Lorsque la meilleure suite de diphones a été sélectionnée, un fichier de sortie, appelé « fichier dipho », est créé. Ce fichier sera analysé par le synthétiseur pour produire la parole de synthèse.

Le format de ce fichier est présenté en Figure 10.6. Les lignes commençant par un point-virgule sont des commentaires. Les autres lignes contiennent chacune les informations relatives à une unité (diphonique) du corpus : le fichier son, le début et la fin du diphone dans ce fichier, le nom et la durée du demi-phone gauche, et le nom et la durée du demi-phone droit. Les positions sont exprimées en ms et les durées, en échantillons.

Les noms et durées des demi-phones servent en fait à articuler en temps réel les lèvres d'un petit visage parlant, appelé « *eFriend* [11] », qui charge également le fichier dipho.

10.1.6.2 Synthèse

Après lecture du fichier dipho, la synthèse extrait les unités acoustiques du corpus de parole, et les concatène en réduisant le traitement du signal au strict minimum.

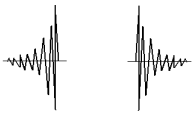
La concaténation de signaux par simple *copier-coller* entraîne de fortes discontinuités du fait de la brusque variation du signal. Afin d'éviter cela, nous appliquons l'algorithme Copy-OLA, acronyme de *Copy, OverLap and Add* (Bozkurt *et al.* 2004). Comme l'illustre la Figure 10.7, le principe général de l'algorithme est de superposer (*overlap*) et d'additionner (*add*) deux signaux au niveau de la dernière période du premier signal et de la première période du second. Cependant, les périodes en question ne sont pas fusionnées telles quelles : la première période subit un fondu décroissant, et la seconde, un fondu croissant. Dans les deux cas, le fondu est réalisé à l'aide d'une demi-fenêtre de Hanning (cf. Equation 10.1.6.3), une forme de fenêtre couramment utilisée en traitement du signal. Copy-OLA est certainement l'algorithme de concaténation *pitch synchrone* le plus simple.

$$F_{Hann}(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{10.1.6.3}$$

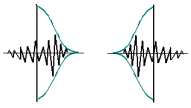
[11] eFriend : www.multitel.be/TTS/layout.php?page=eFriend

```
; Bonjour, je cherche un restaurant.
921.wav   159.500000   363.500000   _   2474   b   790
921.wav   363.500000   477.500000   b   733    o   1145
921.wav   477.500000   614.500000   o   1139   Z   1121
921.wav   614.500000   742.000000   Z   1125   u   1016
921.wav   742.000000   867.500000   u   1009   R   1056
921.wav   867.500000   1139.500000  R   1175   _   3177
531.wav   2281.500000  2457.000000  _   2288   Z   520
531.wav   2457.000000  2509.000000  Z   547    @   349
827.wav   5431.000000  5533.500000  @   619    S   1095
543.wav   5772.687500  5868.812500  S   1055   E   508
543.wav   5868.937500  5916.812500  E   552    R   270
834.wav   3527.000000  3604.500000  R   468    S   882
834.wav   3604.500000  3709.000000  S   809    9   898
555.wav   2575.500000  2645.500000  9   783    R   376
774.wav   5075.000000  5144.000000  R   615    E   580
774.wav   5144.000000  5224.000000  E   578    s   803
774.wav   5224.000000  5315.000000  s   744    t   756
158.wav   4027.937500  4111.187500  t   808    o   615
158.wav   4111.250000  4164.312500  o   565    R   397
177.wav   4200.187500  4306.437500  R   341    a   1434
212.wav   3969.875000  4781.375000  a   1099   _   11885
```

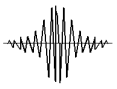
FIG. 10.6: Fichier dipho



(a) Signaux originaux



(b) Demi-Hanning sur la dernière période



(c) Superposition et addition

FIG. 10.7: Schéma du principe Copy-OLA

Comme nous l'avons signalé précédemment, on constate que le signal n'est pas modifié de manière à atteindre une F0 ou une durée données. Le traitement du signal est limité au strict minimum.

10.1.7 Evaluation et analyse

Le corpus que nous utilisons n'est certainement pas le corpus idéal. Il n'a pas été conçu pour LiONS, ne contient qu'une heure de parole et a été prononcé par une locutrice suisse, dont l'accent est perceptible et dont les réalisations prosodiques sont marquées de fortes variations. Nous avons malgré tout fait une évaluation du système, afin de nuancer notre propre analyse.

10.1.7.1 Evaluation

50 auditeurs ont évalué 25 phrases, dont 5 étaient directement extraites du corpus. Le but du test était double :

1. Evaluer l'intelligibilité (*intel*), le naturel de la prosodie (*melo*), la qualité de la concaténation (*concat*) et le confort d'écoute (*comfort*). Chaque critère a été évalué sur une échelle de 1 et 5, 5 étant la meilleure note.
2. Evaluer la différence de qualité entre la voix de synthèse et la voix naturelle.

Les phrases du test peuvent être écoutées sur notre site [12], où les consignes données aux auditeurs et les résultats de l'évaluation phrase par phrase peuvent également être consultés. Les résultats globaux de l'évaluation sont quant à eux présentés en Figure 10.8.

De manière générale, l'évaluation de la parole de synthèse est plutôt positive. La prosodie, la concaténation et le confort d'écoute sont ressentis comme normaux, tandis que l'intelligibilité est fortement mise en avant. Aucun critère ne reçoit une mauvaise note globale.

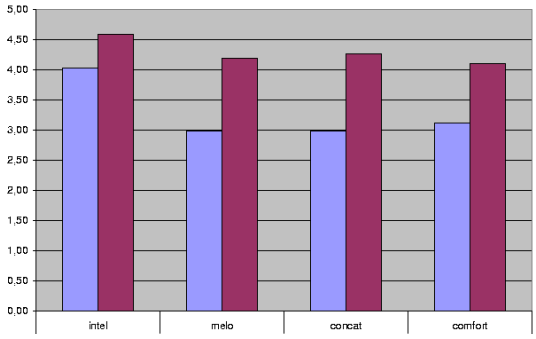
Les résultats concernant la parole naturelle du corpus sont intéressants. Ils montrent le manque de qualité, peut-être subjective, de la voix originale et nous laissent supposer que l'estimation de la qualité de la parole de synthèse s'en ressent.

10.1.7.2 Analyse et propositions

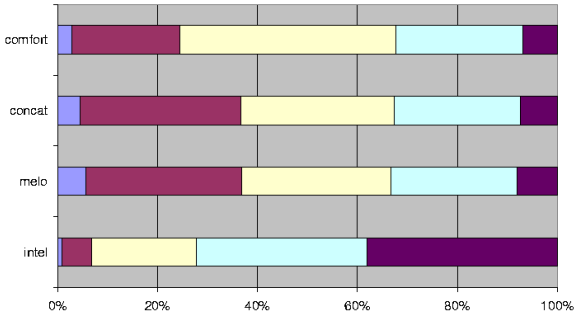
Modélisation de la base. Pour une phrase de la base, la sélection retourne toujours la phrase elle-même. Contrairement à Taylor & Black (1999), nous n'en concluons pas que l'algorithme est particulièrement adapté à la synthèse de phrases d'un domaine particulier.

Quel que soit le corpus et l'algorithme de sélection, la meilleure suite d'unités pour une phrase de la base est logiquement la phrase elle-même, puisque les unités se concatènent parfaitement et que l'étiquetage linguistique est celui recherché par la pré-sélection. Ce résultat assure donc simplement que la base et le système de sélection utilisent le même système d'analyse et que le système de sélection est globalement cohérent.

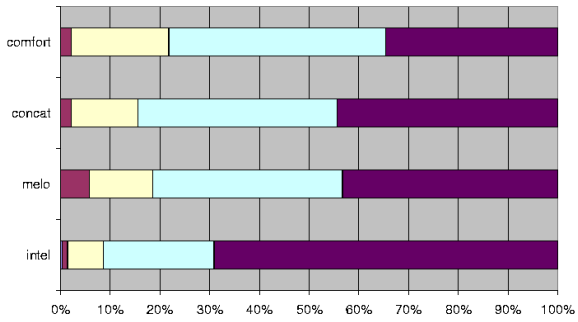
[12] `richardbeaufort.co.nr/phd/2-nuu-LiONS1.0_eval.html`.



(a) Comparaison des 20 phrases de synthèse (bleu) et des 5 phrases du corpus (bordeaux) au niveau de la note moyenne obtenue pour chaque critère d'évaluation



(b) Distribution des notes pour les phrases de synthèse



(c) Distribution des notes pour les phrases du corpus

FIG. 10.8: Evaluation

Qualité de la parole. Dans l'ensemble, la synthèse produite par LiONS est de très bonne qualité. Malgré le manque de couverture du corpus, elle présente très peu de problèmes de concaténation et propose une prosodie tout à fait acceptable. Le coût cible linguistique, le coût de concaténation acoustique et la pondération automatique des critères du coût cible participent à un système équilibré.

Une écoute attentive de la parole générée par cette première version de LiONS révèle cependant quelques caractéristiques gênantes :

1. Un phonème est parfois anormalement long ou court. Or, de nombreux critères suprasegmentaux gèrent la prosodie.
2. Il arrive que la courbe mélodique de fin de phrase ne descende pas, ce qui laisse la phrase en suspens, comme inachevée.
3. Une pause donnée n'est pas réalisée par un silence de longueur constante : pour une même pause, la durée du silence varie parfois fortement. Lorsqu'il y a erreur, le silence est souvent trop long, de sorte qu'une virgule, par exemple, peut être ressentie comme un point.
4. La courbe prosodique globale de la phrase varie parfois de manière inopinée, montant ou descendant excessivement.

Il est donc nécessaire de reconsidérer les critères suprasegmentaux utilisés, les contraintes exprimées sur la fin de phrase et la manière de gérer les pauses.

Temps de sélection. Le système n'est pas temps réel, mais $\frac{1}{4}$ temps réel : 1 seconde de signal demande 4 secondes de traitement. Cependant, nous avons déjà signalé que l'optimisation de la recherche n'était pas l'objectif de ce premier système, qui visait à prouver la possibilité d'obtenir de la parole de synthèse de qualité à l'aide d'un coût cible exclusivement linguistique et d'un coût de concaténation exclusivement acoustique.

Afin d'optimiser le système tout en respectant son algorithme, nous proposons de représenter la totalité du processus de sélection à l'aide de machines à états finis.

Notons cependant que le temps de sélection souffre inutilement d'une double pré-sélection, puisque la pré-sélection des diphones est précédée par la pré-sélection des phones. L'optimisation du temps de sélection nécessite donc, outre une structure adaptée, une remise en cause de l'unité de base de pré-sélection.

Unité de base. Le diphone, utilisé pour la sélection, doit être la seule unité de pré-sélection. Le passage au diphone a en effet une incidence sur le nombre de critères nécessaires à la pré-sélection, comme nous le montrons en Section 10.2 : le fait de rechercher des diphones permet de supprimer les critères segmentaux et de fusionner de nombreux critères suprasegmentaux qui, dans le contexte du diphone, deviennent redondants.

10.2 Révisions et optimisations

Cette section s'ouvre par la présentation des hypothèses propres à cette nouvelle version du système LiONS. Elle présente ensuite la révision des critères de pré-sélection, et se poursuit par la description des modifications apportées au coût de concaténation.

Sur cette base, nous présentons l'optimisation réalisée à l'aide de machines à états finis. Deux points y sont consacrés : le premier décrit le modèle d'optimisation dans ses principes et dans sa globalité, le second détaille l'entraînement des différentes machines du modèle.

Nous présentons ensuite le processus-même de la sélection des unités, et l'étape de synthèse qui la suit. Cette section se conclut par une analyse des améliorations obtenues par la révision des critères et par l'optimisation proposée.

10.2.1 Hypothèses

De l'analyse de la première version de LiONS, nous avons dégagé les hypothèses suivantes, sur lesquelles se fonde notre nouveau système :

Hypothèse 10.2.1 (Sélection par FSMs)**.** *Les machines à états finis constituent un mode de représentation idéal pour optimiser le processus de sélection d'unités non uniformes. Ce mode de représentation permet de précalculer la pré-sélection, le coût cible et le coût de concaténation tout en respectant l'algorithme de sélection initial. Il facilite en outre l'expression de nouvelles contraintes.*

Hypothèse 10.2.2 (Pré-sélection par diphone)**.** *Le diphone est une unité qui condense suffisamment d'informations segmentales pour autoriser la suppression de toutes les informations segmentales de la pré-sélection. La structure du diphone est en outre l'occasion d'exprimer de manière concise de nombreux critères suprasegmentaux.*

Hypothèse 10.2.3 (Durée et pause)**.** *La durée syllabique est avant tout une conséquence du caractère éminemment mécanique de la respiration. Parmi les nombreuses caractéristiques de la respiration, la pause est particulièrement discriminante. En français, la syllabe qui précède directement une pause est toujours allongée, l'importance de son allongement dépendant de la structure interne de la syllabe.*

10.2.2 Révision des critères de pré-sélection

10.2.2.1 Les critères

Nous avons retenu 5 critères linguistiques. Leurs valeurs et leur répartition entre les deux demi-phones et le diphone sont présentées dans la Table 10.6. En tout, la pré-sélection utilise 6 critères $(2+4)$ pour la pré-sélection d'une unité. La Table 10.7 compare les anciens et les nouveaux critères de pré-sélection et, pour les critères communs, met en évidence la différence en terme de nombre de valeurs.

Nous commençons par une vue d'ensemble des critères sélectionnés. Nous expliquons ensuite l'intérêt linguistique du nouveau critère, la distance par rapport à la pause, et les raisons qui justifient l'évolution du nombre de valeurs pour les critères qui étaient déjà utilisés dans le premier système.

Vue d'ensemble. La réduction du nombre de critères retenus est le résultat de deux analyses. La première concerne les poids accordés en moyenne aux différents critères initiaux par le mécanisme de pondération automatique présenté précédemment. Au vu de leurs poids (cf. Annexe 3), certains critères sont inéluctables : l'accent tonique et la longueur de la phrase. Par contre, il est difficile de départager les autres critères, tant leurs pondérations se situent dans un intervalle relativement réduit. Ceci ne signifie cependant pas que le système de pondération soit inopérant, ni que les critères soient dénués de sens. Nous pensons plutôt que certains critères sont redondants, parce qu'ils apportent la même information au système, qui a de ce fait un certain mal à les départager. Ceci signifie que, parmi ces critères, un choix réfléchi est à faire, mais sans l'aide du système de pondération.

La seconde analyse concerne l'intérêt de certains critères dans le cadre du diphone. En effet,

- Si l'on considère qu'un diphone est construit à partir du phone cible et du phone suivant, le contexte droit ne nécessite plus de description.
- Le diphone va de partie stable en partie stable. On peut dès lors considérer que le contexte gauche devient superflu, puisque il est distant du diphone. Nous avons donc décidé de laisser au coût de concaténation le soin d'estimer la qualité du contexte gauche, par le biais de la distance acoustique entre unités voisines.

Nous avons de ce fait supprimé tous les critères segmentaux, puisque ceux-ci ne concernaient que les contextes. En outre, dans le cadre du diphone, nous avons préféré fusionner les critères communs de manière à limiter le nombre de critères nécessaires. Ceci a par ailleurs permis de reconsidérer les valeurs des critères fusionnés.

Distance à la pause. Selon notre Hypothèse 10.2.3, la pause est un critère discriminant dans la définition de la durée d'une syllabe. Ceci repose sur des études phonétiques (Delattre 1959, Crouzet & Angoujard 2006), mais aussi sur nos propres constats faits à l'écoute de parole naturelle : la syllabe a tendance à s'allonger lorsqu'elle précède directement la pause, cet allongement étant particulièrement significatif lorsque la structure syllabique le favorise (cf. Section 10.1.3).

En terme de distance syllabique par rapport à la pause, nous ne nous intéressons qu'à la syllabe finale, *i.e.* celle qui précède *directement* la pause. Les autres syllabes, dans notre modèle, sont indifférenciées. A titre d'exemple, on comparera :

(1) *Il a une vision court terme.*
(2) *J'ai été pris de court.*

	Critère	#	Valeurs	Répartition		
				demi-phone gauche	**diphone**	**demi-phone droit**
1.	Accent tonique	3	`UST`, `SST`, `PST`	v		v
2.	Struct. syllabique	4	`CV`, `CVC`, `VC`, `SNBD`		v	
3.	Pos. Mot/groupe	4	`BEG`, `IN`, `END`, `GBND`		v	
4.	Long. Phrase	3	`SH`, `MD`, `LG`		v	
5.	Pos. Syllabe/pause	4	`1SH`, `1MD`, `1LG`, `2PL`		v	

TAB. 10.6: Critères du coût cible

	Critère	LiONS 1		LiONS 2	
		x	#	x	#
1.	Voisement	2x	2	X	
2.	Type Articulation	2x	3	X	
3.	Lieu Articulation	2x	14	X	
4.	Mode Articulation	2x	5	X	
5.	Labialité	2x	3	X	
6.	Accent tonique	3x	3	2x	3
7.	Struct. syllabique	3x	5	1x	4
8.	Pos./syllabe	1x	3	X	
9.	Long. Syllabe	3x	7	X	
10.	Pos. Syllabe/mot	3x	3	X	
11.	Long. Mot	3x	3	X	
12.	Pos. Mot/groupe	3x	3	1x	4
13.	Groupe	3x	4	X	
14.	Pos. Mot/phrase	3x	3	X	
15.	Long. Phrase	3x	3	1x	3
16.	Pos. Syllabe/pause	X		1x	4

TAB. 10.7: Evolution des critères. Dans chaque cas, le nombre de valeurs (#) et le nombre de parties concernées (x) sont indiqués

où *court* est pénultième en (1) et final en (2). Or, l'allongement ne se produit significativement qu'en (2), alors que dans les deux cas, la structure syllabique est favorable à l'allongement.

Les valeurs de ce critère se déduisent de l'analyse. Nous différencions les syllabes qui se situent devant une pause courte (`1SH`), une pause moyenne (`1MD`) et une pause longue (`1LG`). Par contre, toutes les autres syllabes, dès la pénultième, ne sont plus différenciées (`2PL`).

La distinction entre les trois types de pauses est en outre un guide pertinent de la courbe prosodique de la syllabe, qui ne doit être descendante que dans le cas d'une pause longue.

Accent tonique. Ce critère est le seul qui concerne le demi-phone. La fusion de ce critère en un seul ne se justifiait pas, parce que toutes les combinaisons de valeurs sont *a priori* autorisées et ne peuvent être exprimées de manière concise.

Structure syllabique. Du fait de la fusion du critère au niveau du diphone, deux valeurs utilisées dans LiONS 1 ont été supprimées (V et silence) et une nouvelle valeur est apparue : `SBND` [13], pour la césure. L'idée est de n'attribuer une valeur syllabique différente de *césure* que si les deux phones appartiennent à la même syllabe. La valeur *césure* est utile pour permettre d'exprimer de manière *concise* que deux phones se situent en césure syllabique, quelles que soient les structures des syllabes auxquelles ils appartiennent (V/CV,VC/CVC,CV/VC, etc.). La suppression des anciennes valeurs est due au fait que :

- Le *silence* est une syllabe à lui seul. Dans le contexte du diphone, il est donc systématiquement en césure.
- La syllabe de type V indique initialement une syllabe constituée d'une simple voyelle à laquelle une semi-voyelle satellite pouvait se greffer. Cependant, le caractère « semi-consonantique » d'une semi-voyelle nous a fait préférer l'assimiler à une consonne et considérer qu'elle constitue avec la voyelle une syllabe CV lorsqu'elle précède la voyelle, et VC lorsqu'elle la suit. La syllabe de type V est donc systématiquement en césure dans le diphone.

Le mot par rapport au groupe. Un raisonnement semblable à celui développé ci-dessus est à l'origine de la valeur `GBND` [14], pour « frontière », ajoutée à la position du mot par rapport au groupe : nous n'attribuons une valeur différente de *frontière* que si les deux demi-phones appartiennent au même groupe.

[13] `SBND` signifie *syllable boundary*.
[14] `GBND` signifie *group boundary*.

10.2.2.2 La pondération

Nous avons simplement exécuté le système automatique décrit en Section 10.1.5 sur le nouveau jeu de critères du coût cible.

10.2.3 Révision des critères du coût de concaténation

Les deux révisions principales sont la définition de valeurs permettant la normalisation des critères du coût de concaténation, et le choix d'un critère supplémentaire pour améliorer le coût dans sa globalité. La normalisation a en outre été l'occasion de proposer de nouveaux pondérateurs, plus faciles à interpréter.

10.2.3.1 Normalisation des critères

En statistique, la normalisation consiste à projeter les valeurs d'un critère sur une échelle bornée. Typiquement, pour un ensemble de valeurs ≥ 0, cette échelle est bornée entre 0 et 1. La normalisation offre entre autres l'avantage de faciliter la comparaison de plusieurs critères qui, sans normalisation, évoluent sur des échelles différentes.

Les valeurs des critères du coût cible sont par nature normalisés, étant donné que les critères établissent des distances que nous avons par défaut échelonnées entre 0 et 1.

Par contre, comme nous l'avons détaillé en Section 10.1.5, les valeurs des critères du coût de concaténation ne sont pas normalisées. Ceci engendre trois inconvénients :

1. Il n'est pas possible de comparer ces valeurs.
2. Comme nous l'avons signalé dans la même Section, il est difficile d'estimer dans quelle mesure les pondérateurs choisis accordent plus d'importance à l'un ou à l'autre des critères, tant leurs échelles de valeurs divergent.
3. Il est difficile d'évaluer l'importance respective du coût cible par rapport au coût de concaténation, puisque les deux coûts évoluent sur des échelles fortement différentes.

Afin de résoudre ce problème, nous avons normalisé les valeurs des critères du coût de concaténation.

Normalisation par phonème. Les valeurs des critères du coût de concaténation (F0, énergie, spectre et durée) dépendent fortement du phonème. Par exemple, une consonne sourde aura toujours une F0 nulle, alors qu'une voyelle devrait avoir une courbe de F0 variée. La normalisation, pour respecter les particularités de chaque phonème, doit dès lors être réalisée par phonème.

Borne de normalisation. Afin de borner les valeurs d'un critère entre 0 et 1, une normalisation simple consisterait à mémoriser la valeur maximale max de ce critère dans le corpus, afin d'obtenir la valeur normée d'une valeur v donnée comme suit :

$$\texttt{Norm}(v) = \frac{v}{max} \tag{10.2.3.1}$$

qui, pour un ensemble de valeurs ≥ 0, sera par définition toujours située entre 0 et 1. Cependant, les valeurs prélevées sur le corpus peuvent dans certains cas être aberrantes, puisqu'elles dépendent d'informations parfois erronées. Par exemple, un signal peut être étiqueté [s], mais être en réalité un [z]. Dans ce cas, la F0 sera probablement positive, alors qu'elle devrait être nulle. La valeur maximale d'un critère n'est donc pas une information pertinente.

Afin d'exclure les valeurs aberrantes, nous avons choisi de poser comme borne supérieure la somme de la médiane μ et de l'écart type σ :

$$\texttt{Norm}(v) = \frac{v}{\mu + \sigma} \tag{10.2.3.2}$$

où

$$\mu = \begin{cases} v_{\frac{n+1}{2}} & \text{si } n \text{ est impair,} \\ \\ \frac{1}{2}\left(v_{\frac{n}{2}} + v_{\frac{n+1}{2}}\right) & \text{sinon} \end{cases} \tag{10.2.3.3}$$

et

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2} \tag{10.2.3.4}$$

La définition de cette borne nécessite trois remarques :

1. Nous proposons une borne calculée sur la médiane, afin d'éviter la moyenne $\overline{v}$, dont le calcul inclut les valeurs aberrantes :

$$\overline{v} = \frac{1}{n}\sum_{i=1}^{n} v_i \tag{10.2.3.5}$$

2. De ce fait, par souci de cohérence, l'écart type est lui-même calculé en fonction de la médiane et non de la moyenne.
3. Dans le cas où v est une valeur aberrante, $\texttt{Norm}(v)$ sera supérieure à 1. Ceci permet de défavoriser explicitement ces valeurs, sans pour autant les exclure *a priori*.

10.2.3.2 Ajout d'un critère

Nous avons constaté que certaines réalisations acoustiques dans le corpus ont une durée exagérément brève ou longue. Il est donc préférable que le système de sélection les évite, mais sans les exclure *a priori*.

Dans ce but, nous avons ajouté un second critère relatif à la durée : le *rapport de durée RD*, que nous définissons comme suit :

$$\texttt{RD}(d) = \begin{cases} 0 & \text{si } d_{min} \leq d \leq d_{max} \\ \\ \frac{\mu + \text{abs}(\mu - d)}{\mu} & \text{sinon} \end{cases} \tag{10.2.3.6}$$

où $d_{min} = (\mu - \sigma)$ et $d_{max} = (\mu + \sigma)$. En somme, le rapport de durée soit vaut 0, soit est supérieur à 1. De la sorte, les durées bornées ne sont pas distinguées entre elles, mais par contre se distinguent nettement des durées hors bornes.

Etant donné que le critère évalue la distance sur les parties contiguës de deux diphones |ba| et |ac|, la distance vaut :

$$D_{RD}(|\text{ba}|, |\text{ac}|) = \frac{1}{2}\texttt{RD}(\texttt{Right}(|\text{ba}|)) + \frac{1}{2}\texttt{RD}(\texttt{Left}(|\text{ac}|)) \tag{10.2.3.7}$$

10.2.3.3 Pondération des critères

Les pondérateurs ont été déterminés *manuellement* en accordant un poids global identique au coût cible et au coût de concaténation. De nombreuses pondérations ont été testées. Le jeu de pondérateurs retenu est présenté en Table 10.8.

Critères	**Poids**
F0	20
Spectre	10
Energie	5
Durée	2
Rapport de durée	1

TAB. 10.8: Poids des critères du coût de concaténation

On constate que la F0 est prépondérante, suivie par le spectre. Il est intéressant de souligner que les trois autres paramètres, quoique moins pondérés, améliorent cependant nettement les résultats. Le dernier critère par exemple, le rapport de durée, intervient pour peu dans le coût global, mais permet d'éviter certains écueils qui étaient fréquents lorsque ce critère n'était pas utilisé. Par contre, mettre trop l'accent sur ces trois critères mineurs est directement sanctionné par de franches discontinuités au niveau de la F0 ou du spectre. De même, donner autant d'importance au spectre qu'à la F0 a tendance à masquer l'apport des trois dernier critères, et inverser les pondérations de la F0 et du spectre ne permet pas d'éviter de franches discontinuités de la F0.

10.2.4 Modèle d'optimisation

Nous avons posé dans notre Hypothèse 10.2.1, que la pré-sélection des candidats et le calcul des deux coûts (cible – concaténation) peuvent être pré-calculés et représentés sous la forme de FSMs. Or, au contraire de Allauzen *et al.* (2004), nous ne proposons pas un modèle de langue.

10.2.4.1 Diviser pour mieux régner

Nous appliquons ici notre Hypothèse 1, afin d'alléger la construction du système de sélection.

Séparation cible – concaténation. Contrairement à Bulyko & Ostendorf (2001), nous estimons que la phase de pré-sélection et la phase de sélection doivent être représentées par des machines séparées. Cette approche permet de respecter l'algorithme initial. Elle facilite également la construction des machines, étant donné que les problèmes sont pensés et gérés séparément. Il faut donc au moins 2 machines :

1. Une machine pour le coût cible.
2. Une machine pour le coût de concaténation.

Subdivision du coût cible. Quel que soit le corpus de parole, son analyse révèle quatre types d'unités :

1. Les *diphones présents*. Ce sont des diphones qui, sans considérer les critères de pré-sélection, ont au moins 1 représentant dans le corpus.
2. Les *diphones manquants*. Ce sont des diphones qui, sans considérer les critères de pré-sélection, sont complètement absents du corpus. Notre corpus, par exemple, ne contient aucune occurrence de |œ̃ œ|, ni de |ɛ̃ œ|.
3. Les *cibles présentes*. Ce sont des diphones accompagnés de valeurs précises pour les critères retenus qui ont *au moins n* réalisations acoustiques dans le corpus.
4. Les *cibles manquantes*. Ce sont des diphones accompagnés de valeurs précises pour les critères retenus qui ne proposent pas *au moins n* réalisations acoustiques dans le corpus. Certaines cibles manquantes n'ont aucune réalisation acoustique. C'est le cas, par exemple, de

|ɛ e|	0	1	CV	début	long	1-court
	Diph.	*Acc.*	*Acc.*	*Syl.*	*Pos./gr.*	*Lg. Phr.*	*Pos./pau.*

La modélisation d'une machine qui gérerait la totalité de ces unités serait inutilement complexe. Nous préférons subdiviser la pré-sélection en étapes successives :

1. S'agit-il d'un diphone manquant ? Oui → Substitution.
2. S'agit-il d'une cible manquante ? Oui → Substitution pondérée.
3. Pour une cible présente, obtention des occurrences du corpus.

10.2.4.2 Vue globale des FSMs

L'idée est de gérer les différentes unités du coût cible et le coût de concaténation à l'aide de 4 machines :

1. Une transducteur T_{DM} gère les diphones manquants, auxquels il substitue des diphones présents. Par exemple, |ɛ̃ œ| pourra être projeté sur le Diphone Existant |ɛ œ|. Ces substitutions pallient un manque qui ne devrait pas être. Elles permettent simplement au processus de synthèse de se terminer, quoi qu'il arrive. Nous considérons donc qu'elles ne doivent pas être pondérées. La Figure 10.9 (a) illustre ce transducteur.
2. Un transducteur T_{CM} gère les cibles manquantes, auxquelles il substitue des cibles présentes. Ces substitutions sont pondérées par le coût cible, puisque la machine propose des substitutions de valeurs pour certains critères d'une cible. La Figure 10.9 (b) illustre ce transducteur.
3. Un transducteur T_{CP} gère les cibles présentes. Pour chaque cible présente, T_{CP} recense les occurrences du corpus. Ces occurrences sont représentées par les identifiants des réalisations acoustiques dans le corpus. Par exemple, la cible présente [|ɛ e|, 1, 0, césure, début, long, 2-plus] correspond dans notre corpus aux identifiants {123, 2 510, 4 576, 30 027}. Ce transducteur, illustré en Figure 10.10 (a), n'est pas pondéré puisque cible et identifiants se correspondent parfaitement.
4. Un automate A_{CO} gère le coût de concaténation, illustré par la Figure 10.10 (b). Dans cette machine, l'état initial est 0 et représente le début d'une phrase. Tout état i avec $i > 0$ représente le diphone portant l'identifiant i dans le corpus. L'état initial possède des transitions vers les identifiants qui peuvent commencer une phrase. Aucune transition, par contre, n'entre dans l'état initial. Toute transition d'un état i vers un état j représente le coût de concaténation calculé entre la frontière droite du diphone i et la frontière gauche du diphone j. Les états correspondant à des identifiants qui peuvent terminer la phrase sont finaux.

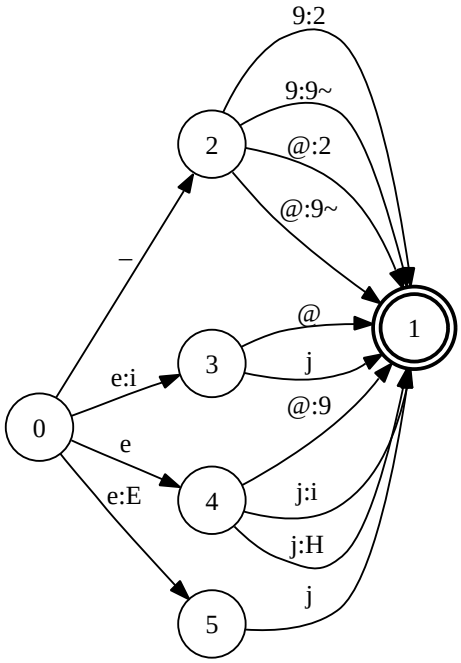
10.2.5 Entraînement adapté au modèle

Les premières étapes du nouvel entraînement sont les mêmes que celles réalisées par l'entraînement de la première version de LiONS (cf. Section 10.1.4) :

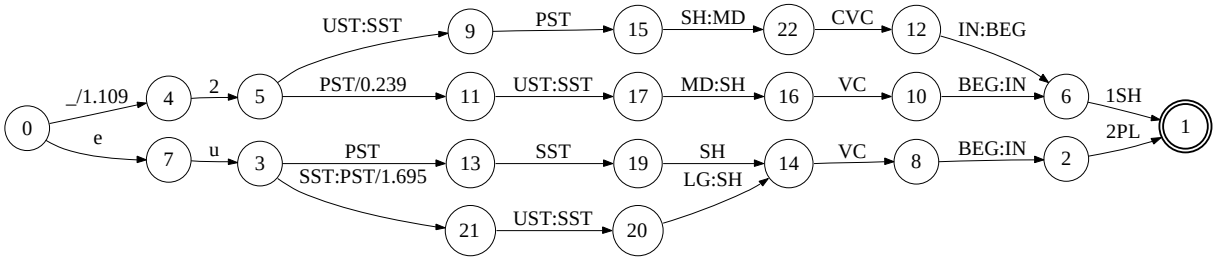
1. L'alignement phonétique et l'analyse acoustique.
2. La création de la table d'étiquetage phonétique.
3. L'entraînement de la pondération phonétique.

A ces étapes s'ajoutent :

1. La construction de T_{DM}, le FST des diphones manquants.
2. La construction de T_{CM}, le WFST des cibles manquantes.
3. La construction de T_{CP}, le FST des cibles présentes.
4. La construction de A_{CO}, le WFSA de concaténation.

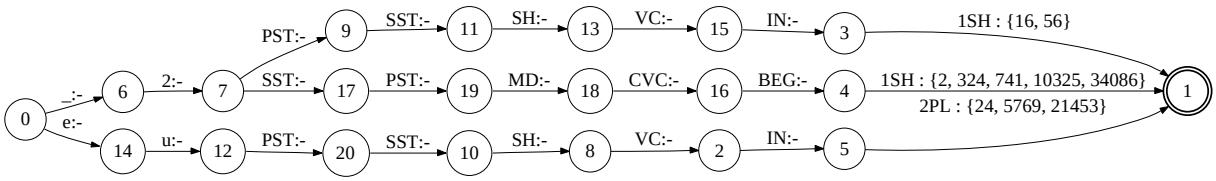


(a) T_{DM}. Les diphones manquants illustrés sont |_œ|, |_ə|, |eə| et |ej|

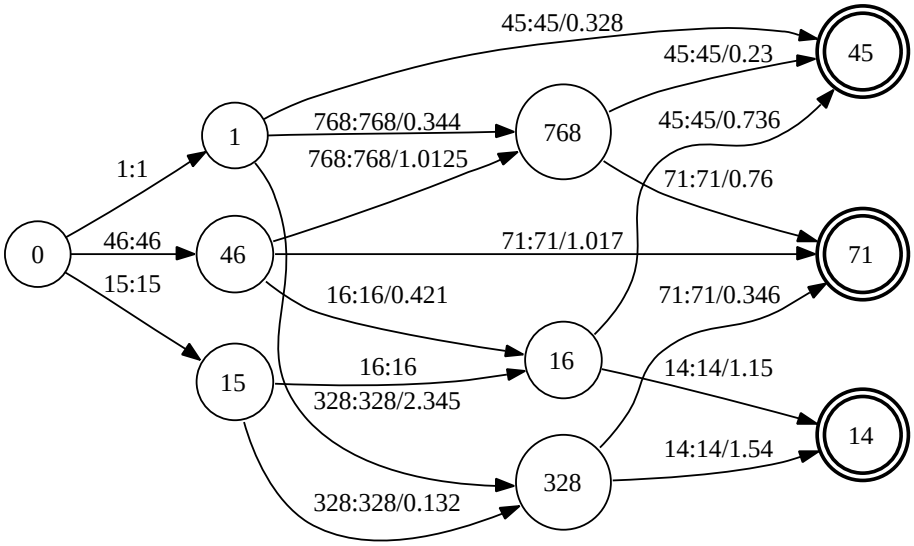


(b) T_{CM}. La machine est pondérée par le coût cible

FIG. 10.9: LiONS 2 : vue globale des FSMs, partie 1



(a) T_{CP}. Les identifiants se trouvent sur la dernière transition de chaque diphone présent, et ont été rassemblés entre accolades pour plus de lisibilité



(b) A_{CO}. Le coût de concaténation est indiqué si > 0

FIG. 10.10: LiONS 2 : vue globale des FSMs, partie 2

Phoneme	**classe**	**lieu**	**Art.**	**Labialité**	**Vois.**	**Aperture**
_	indéfini	indéfini	indéfini	indéfini	indéfini	indéfini
i	voyelle	palatal	oral	étiré	voisé	1
e	voyelle	palatal	oral	étiré	voisé	2
ɛ	voyelle	palatal	oral	étiré	voisé	3
ɛ̃	voyelle	palatal	nasal	étiré	voisé	3
a	voyelle	palatal	oral	étiré	voisé	4
y	voyelle	postpalatal	oral	arrondi	voisé	1
ø	voyelle	postpalatal	oral	arrondi	voisé	2
œ	voyelle	postpalatal	oral	arrondi	voisé	3
œ̃	voyelle	postpalatal	nasal	arrondi	voisé	3
ə	voyelle	postpalatal	oral	arrondi	voisé	4
u	voyelle	vélaire	oral	arrondi	voisé	1
o	voyelle	vélaire	oral	arrondi	voisé	2
ɔ	voyelle	vélaire	oral	arrondi	voisé	3
ɔ̃	voyelle	vélaire	nasal	arrondi	voisé	3
ɑ̃	voyelle	vélaire	nasal	arrondi	voisé	4
w	semi-voy.	vélaire	oral	arrondi	voisé	1
ɥ	semi-voy.	postpalatal	oral	arrondi	voisé	1
j	semi-voy.	palatal	oral	étiré	voisé	1
p	consonne	bilabial	plosif	indéfini	non-voisé	indéfini
b	consonne	bilabial	plosif	indéfini	voisé	indéfini
m	consonne	bilabial	nasal	indéfini	voisé	indéfini
f	consonne	labiodental	constrictif	indéfini	non-voisé	indéfini
v	consonne	labiodental	constrictif	indéfini	voisé	indéfini
t	consonne	alvéodental	plosif	indéfini	non-voisé	indéfini
d	consonne	alvéodental	plosif	indéfini	voisé	indéfini
n	consonne	alvéodental	nasal	indéfini	voisé	indéfini
s	consonne	alvéolaire	constrictif	indéfini	non-voisé	indéfini
z	consonne	alvéolaire	constrictif	indéfini	voisé	indéfini
ʃ	consonne	postalvéol.	constrictif	indéfini	non-voisé	indéfini
ʒ	consonne	postalvéol.	constrictif	indéfini	voisé	indéfini
k	consonne	vélaire	plosif	indéfini	non-voisé	indéfini
g	consonne	vélaire	plosif	indéfini	voisé	indéfini
ɲ	consonne	palatal	nasal	indéfini	voisé	indéfini
ŋ	consonne	vélaire	nasal	indéfini	voisé	indéfini
ʁ	consonne	postvélaire	liquide	indéfini	voisé	indéfini
l	consonne	latéral	liquide	indéfini	voisé	indéfini

Tab. 10.9: Critères articulatoires toutes classes confondues

10.2.5.1 Construction de T_{DM}

La première étape de l'algorithme consiste à établir, au travers d'une double itération sur les phonèmes, la liste de tous les diphones manquants du corpus [15].

Pour chaque diphone manquant, nous proposons ensuite une liste de substitutions, établie selon trois critères :

1. Toute substitution doit être un diphone présent...
2. La somme des réalisations acoustiques des membres de la liste de substitutions doit couvrir au moins n réalisations acoustiques. Nous avons fixé n à 40.
3. Les substitutions sont choisies en fonction de leur proximité avec le diphone manquant.

La liste comporte donc les m meilleures substitutions existantes qui, *ensemble*, couvrent au moins n réalisations dans le corpus. L'algorithme de construction de la liste de substitutions pour un diphone manquant donné est présenté en Pseudocode 26.

Distance entre diphones. Le cœur de l'algorithme est évidemment la fonction `DistanceDM` (ligne 2), qui évalue la distance entre chaque diphone présent du vecteur $\overrightarrow{DP}$ et le diphone manquant DM, et classe $\overrightarrow{DP}$ en fonction.

La distance entre deux diphones d_1 et d_2 est la somme de la distance entre leurs demi-phones gauches et de la distance entre leurs demi-phones droits :

$$D(d_1, d_2) = D(\texttt{Left}(d_1, d_2)) + D(\texttt{Right}(d_1, d_2)) \tag{10.2.5.1}$$

La distance entre deux demi-phones dm_1 et dm_2 est quant à elle calculée en termes articulatoires, comme la somme des distances entre les valeurs des critères pour chaque demi-phone :

$$D(dm_1, dm_2) = \sum_{i=1}^{n} D(F_i(dm_1), F_i(dm_2)) \tag{10.2.5.2}$$

où $F_i(x)$ retourne la valeur du i^e critère de x.

Les critères articulatoires sont *a priori* ceux présentés en Section 8.3.1, dans les Tables 8.2, 8.3 et 8.4. Cependant, ces critères n'autorisent pas l'établissement d'une distance entre classes (voyelles, consonnes, semi-voyelles) parce qu'ils diffèrent d'une classe à l'autre. Pour cette raison, nous avons condensé les descriptions articulatoires de l'ensemble des phonèmes en une seule liste de critères. La Table 10.9 présente la liste établie pour le français.

Quel que soit le critère, la distance est toujours bornée entre 0 et 1. Pour la plupart des critères, une différence de valeur implique une distance de 1. C'est le cas, par exemple, de la classe et du voisement.

Deux critères, par contre, permettent un échelonnement des distances entre 0 et 1. Il s'agit du lieu d'articulation et de l'aperture, dont les valeurs ne sont objectivement pas

[15] A l'exception du diphone exclusivement fait de silences, |_-_|, que nous n'acceptons pas.

équidistantes. Par exemple, la distance entre les lieux d'articulation *bilabial* et *labiodental*, qui sont côte à côte, est objectivement plus petite qu'entre *bilabial* et *vélaire*, qui sont aux deux extrémités de la cavité buccale. Dans ce cas, la distance entre deux valeurs est le rapport entre le nombre de positions séparant les deux valeurs et le nombre n de valeurs du critères :

$$D(v_1, v_2) = \frac{\text{abs}(pos(v_1) - pos(v_2))}{n} \tag{10.2.5.3}$$

où $pos(x)$ définit la position de x dans l'échelle des valeurs.

Compilation de T_{DM}. A partir des listes de substitutions, nous créons un fichier respectant la syntaxe de notre compilateur de règles de réécriture, Ovide (cf. Section 6.2). Ce fichier, dont la structure est présentée en Figure 10.11, limite tout d'abord le langage autorisé (`LANGIN`) à la liste des diphones manquants. Il présente ensuite les substitutions autorisées sous la forme de règles de réécriture fort simples (`RULE`). Le transducteur T_{DM} compilé par Ovide est de la forme de celui présenté en Figure 10.9 (a) et prend 13,4 Ko au format binaire.

10.2.5.2 Construction de T_{CM}

Le principe de la construction de T_{CM} est relativement similaire à celui de T_{DM}. Pour une cible manquante, il s'agit de déterminer les m cibles présentes les plus proches qui couvrent au moins n réalisations acoustiques dans le corpus. Ici également, n vaut 40. Il est important de rappeler que parmi ce que nous qualifions de *cibles manquantes* figurent des cibles qui proposent moins de 40 réalisations dans le corpus.

Pour une cible manquante donnée, la construction d'une liste de substitutions est exclusivement réalisée parmi les réalisations acoustiques qui appartiennent au même diphone. L'algorithme est présenté en Pseudocode 27.

Distance entre cibles. Le cœur de l'algorithme est la fonction `DistanceCM` (ligne 2), qui évalue la distance entre chaque cible présente du vecteur $\overrightarrow{CP}$ et la cible manquante CM, et classe $\overrightarrow{CP}$ en fonction.

La distance entre deux cibles c_1 et c_2 est évidemment établie à l'aide du coût cible lui-même, tel qu'il était réalisé au cours du processus de sélection dans la première version de LiONS (cf. 10.1.6). Les cibles étant de type diphonique, le coût est établi à l'aide de l'Equation 10.1.6.1. Ceci assure que le coût cible est calculé de manière identique à l'algorithme original et permet de pré-calculer l'ensemble des coûts cibles.

Ce coût cible diffère uniquement du précédent dans le sens où il emploie la liste de critères révisée, présentée en Section 10.2.2.

Require: Un diphone manquant DM,
Le vecteur des diphones présents $\overrightarrow{DP}$,
Le nombre de réalisations désirées n
Ensure: Une liste de substitutions L couvrant au moins n réalisations du corpus
1: $L \leftarrow \emptyset$
2: `DistanceDM`(DM,$\overrightarrow{DP}$)
3: $m \leftarrow 0$
4: $i \leftarrow 0$
5: **while** $i < |\overrightarrow{DP}|$ **and** $m < n$ **do**
6: $L \leftarrow L \cup \overrightarrow{DP}[i]$
7: $m \leftarrow m + |\overrightarrow{DP}[i]|$
8: $i \leftarrow i + 1$
9: **end while**
10: **return** L

Pseudocode 26: Liste de substitutions pour un diphone manquant

Require: Une cible manquante CM,
Le vecteur des cibles présentes $\overrightarrow{CP}$ limité aux réalisations du diphone,
Le nombre de réalisations désirées n
Ensure: Une liste de substitutions L couvrant au moins n réalisations du corpus
1: $L \leftarrow \emptyset$
2: `DistanceCM`(CM,$\overrightarrow{CP}$)
3: $m \leftarrow 0$
4: $i \leftarrow 0$
5: **while** $i < |\overrightarrow{CP}|$ **and** $m < n$ **do**
6: $L \leftarrow L \cup \overrightarrow{CP}[i]$
7: $m \leftarrow m + |\overrightarrow{CP}[i]|$
8: $i \leftarrow i + 1$
9: **end while**
10: **return** L

Pseudocode 27: Liste de substitutions pour une cible manquante

Compilation de T_{CM}. Comme dans le cas de T_{DM}, nous créons pour T_{CM} un fichier au format « Ovide ». La structure de ce fichier, présentée en Figure 10.12, est fort comparable à celle du fichier créé pour T_{DM}. Deux différences, cependant :

1. On constate que certaines règles sont optionnelles (?→) tandis que d'autres sont obligatoires (→). Le principe est le suivant :
   - Si la cible manquante n'existe pas du tout dans le corpus, la dernière règle qui la concerne est obligatoire, afin de supprimer la Cible de la liste de substitutions.
   - Si la cible manquante existe dans le corpus, toutes les règles qui la concerne sont optionnelles, afin de conserver la Cible dans la liste de substitutions. Ceci signifie que l'un des chemins du transducteur ne réécrira pas la Cible et ne la pondérera pas non plus. Dans ce cas, la Cible reste donc son propre meilleur candidat.
2. Les règles sont pondérées par le coût cible, qui sera intégré au transducteur.

Le transducteur T_{CM} compilé par Ovide est de la forme de celui présenté en Figure 10.9 (b) et prend 1,8 Mo au format binaire.

10.2.5.3 Construction de T_{CP}

A partir de la table d'étiquetage, nous créons directement un fichier « Ovide » (cf. Figure 10.13) dont les règles permettent de projeter chaque cible présente sur l'ensemble de ses réalisations acoustiques, représentées sous la forme de leurs identifiants. Le transducteur T_{CP} compilé par Ovide est de la forme de celui présenté en Figure 10.10 (a) et prend 1,3 Mo au format binaire.

10.2.5.4 Construction de A_{CO}

Le dernier FSM à construire est un WFSA représentant les coûts de concaténation. Le principe de base est donc de précalculer la totalité des coûts de concaténation possibles, et de les mémoriser dans l'automate.

Or, comme l'ont relevé à juste titre Beutnagel *et al.* (1999b), il n'est pas réaliste de mémoriser la totalité des coûts de concaténation, quelle que soit la structure utilisée. Le constat de ces chercheurs est d'autant plus justifié que leur corpus compte 84 000 demi-phones qui autorisent 1,76 milliards de concaténations.

Notre corpus compte un peu plus de 37 000 diphones, qui autorisent « seulement » 88 191 926 concaténations. Il n'empêche, représenter l'ensemble de ces concaténations à l'aide d'un WFSA au format de notre bibliothèque (cf. Section **??**) représenterait approximativement 1,1 Go tant sur le disque qu'en mémoire. Autant dire que ce n'est pas réaliste, et qu'un élagage a été nécessaire.

Critère d'élagage. Le principe d'élagage est évidemment de conserver les meilleures concaténations en terme de coût de concaténation (cf. Section 10.2.3).

La méthode d'élagage repose quant à elle sur la définition de nouvelles contraintes.

```
[LANGIN] # les diphones manquants
  _ 9 | _ @ | e @ | e j | ...
[RULE] # les substitutions pour chaque diphone manquant
  _ 9 → _ 2 | _ 9
  _ @ → _ 2 | _ 9
  e @ → i @ | e 9
  e j → i j | e i | e H | E j
  ...
```

FIG. 10.11: Fichier Ovide pour T_{DM}. Le caractère « # » marque le début d'un commentaire

```
[LANGIN] # les cibles manquantes
  _ 2 UST PST SH CVC IN 1SH
  e u SST UST LG VC BEG 2PL
  e u PST SST SH VC BEG 2PL
  ...
[RULE] # les substitutions proposees
  _ 2 UST PST SH CVC IN 1SH ?→ _ 2 SST PST MD CVC BEG 1SH / 1.435
  _ 2 UST PST SH CVC IN 1SH → _ 2 SST PST MD CVC BEG 2PL / 1.726
  e u SST UST LG VC BEG 2PL → e u PST SST SH VC IN 2PL / 2.021
  e u PST SST SH VC BEG 2PL → e u PST SST SH VC IN 2PL / 0.326
  ...
```

FIG. 10.12: Fichier Ovide pour T_{CM}

```
[LANGIN] # les cibles presentes
  _ 2 SST PST MD CVC BEG 1SH
  _ 2 PST SST SH VC IN 1SH
  e u PST SST SH VC IN 2PL
  ...
[RULE] # projection des cibles presentes sur les identifiants
  1SH → 2 | 324 | 741 | 10325 | 34086 :: 2 SST PST MD CVC BEG _
  1SH → 16 | 56 :: 2 PST SST SH VC IN _
  2PL → 24 | 5769 | 21453 :: e u PST SST SH VC IN _
  ...
```

FIG. 10.13: Fichier Ovide pour T_{CP}

Nouvelles contraintes. Notre représentation sous la forme d'un WFSA se base sur le postulat suivant :

Postulat 10.2.1 (Contraintes de concaténation)**.** *La structure de représentation ne peut modifier l'algorithme de sélection de manière involontaire. Elle peut par contre être l'occasion d'ajouter des contraintes supplémentaires qui améliorent l'algorithme.*

dont nous dégageons l'hypothèse suivante :

Hypothèse 10.2.4 (Contraintes de concaténation)**.** *Une machine à états finis pondérée permet de respecter l'algorithme de sélection, tout en refusant certaines transitions nuisibles à la qualité globale de la sélection.*

Sur la base de cette hypothèse, nous avons défini les contraintes suivantes sur le WFSA de concaténation, de manière à maîtriser l'évolution de la courbe prosodique :

1. *Répartition équilibrée.* L'objectif est d'obtenir une répartition équilibrée des concaténations possibles entre les différentes unités d'un diphone donné dans le corpus. Ceci signifie que nous ne retenons pas les meilleures concaténations de manière globale, mais les meilleures concaténations *par unité*. La nuance est importante : nous contraignons le processus de sélection à ne pas favoriser systématiquement certaines unités qui, au vu de leurs caractéristiques acoustiques, accapareraient une majeure partie des concaténations retenues. Ce faisant, nous modifions l'algorithme de sélection. Cependant, cette modification n'est pas dictée par la structure de représentation, mais par un souhait de rééquilibrer le processus de programmation dynamique.
2. *Gestion du début de phrase.* Nous avons mentionné précédemment que chaque état correspond à une unité du corpus, à l'exception de l'état 0 qui est l'état initial de l'automate et représente le début d'une phrase. De l'état 0, nous n'acceptons des transitions que vers les états correspondant à des unités dont le premier demi-phone est le silence : |_.| [16].
3. *Gestion de la fin de phrase.* Afin de forcer la courbe intonative à descendre en fin de phrase, les seuls états finaux de l'automate sont ceux qui correspondent à des unités situées en fin de phrase dans le corpus. Il s'agit évidemment d'unités de la forme |. _|. Malheureusement, du fait de la mauvaise couverture du corpus, certains diphones |. _| ne se produisent jamais en fin de phrase. C'est le cas, par exemple, du diphone |œ̃ _|. Dans ce cas, nous acceptons que *toutes* les unités du diphone concerné terminent la phrase en rendant finaux les états correspondants.
4. *Gestion des pauses à l'intérieur de la phrase.* Afin d'éviter une courbe intonative de fin de phrase à l'intérieur de la phrase, les états correspondant à des unités situées en fin de phrase dans le corpus n'acceptent aucune transition sortante. Ces unités ne peuvent donc être que finales. Par contre, les états correspondant à des unités situées à

[16] Le point signifie « n'importe quel demi-phone ».

une pause intérieure (courte ou moyenne) acceptent des transitions vers tous les états correspondant à des unités situées à une pause intérieure (courte ou moyenne).

Elagage. Fixer le nombre maximum de concaténations par unité vers chaque diphone revient à déterminer un seuil. Or, les seuils sont souvent arbitraires. Afin de limiter l'arbitraire autant que faire se peut, nous avons synthétisé un ensemble de phrases à l'aide de différentes machines de concaténation, en retenant de 20 à 4 concaténations par unité vers chaque diphone. L'analyse des résultats à montré que les unités sélectionnées restent identiques tant que l'on ne descend pas en-dessous de 8 concaténations par unité vers chaque diphone.

Les seules unités qui acceptent plus de 8 transitions sont celles dont le second demiphone est le silence et qui correspondent à des pauses situées à l'intérieur de la phrase. Dans ce cas, toutes les transitions vers des unités commençant par un silence situé à l'intérieur de la phrase sont acceptées, sans restriction.

Représentation compacte. Comme nous l'avons détaillé en Section 6.1.4, les transitions de nos FSMs sont constituées de 4 informations : un symbole d'entrée, un symbole de sortie, un poids et un état suivant. Or, en ne conservant que 8 concaténations, l'automate A_{CO} prend 159 Mo lorsqu'il est représenté au format binaire, qu'il soit minimisé ou non. La minimisation ne modifie pas fondamentalement la taille de la machine, parce que chaque état de l'automate correspond à une unité particulière et que les concaténations conservées diffèrent d'une unité à l'autre. Seuls les états finaux dont aucune transition ne sort peuvent être fusionnés par la minimisation.

Pour autant que la mémoire vive de l'ordinateur soit suffisante (au moins 256 Mo), la taille de l'automate ne ralentit pas le processus : le format binaire assure un chargement rapide (cf. Section 6.1.5) et la sélection n'est en rien ralentie, comme nous le détaillons en Section 10.2.7. Cependant, le développement d'un outil informatique doit aussi veiller à minimiser la quantité de données requises, ne serait-ce que pour en faciliter la distribution ou pour lui permettre de cohabiter en mémoire vive avec d'autres programmes utilisés simultanément.

Une caractéristique de l'automate A_{CO} peut être exploitée de manière à réduire la taille de la machine. Comme l'illustre la Figure 10.10 (b), toutes les transitions de nos machines présentent systématiquement une valeur identique pour le symbole d'entrée, le symbole de sortie et l'état suivant. Ceci est logique, puisque chaque état représente une unité, et que les transitions entre unités représentent le coût de concaténation entre ces unités. Il y a donc redondance entre ces trois informations.

Ceci nous a conduit a définir une nouveau type de machine dans notre bibliothèque : le *graphe*, que nous avons présenté en Section 6.1.8.2. Dans le principe, nous rappelons qu'un graphe de notre bibliothèque est une machine dont toutes les transitions sont exclusivement constituées de 2 informations : un poids et un état suivant. Le graphe économise donc 2x4 octets par transition. La Figure 10.14 illustre la simplification obtenue, qui permet à A_{CO}

de passer de 159 Mo à 79,7 Mo, sans perte d'information. Le processus peut dès lors être exécuté, sans ralentir, sur un ordinateur pourvu de 128 Mo de mémoire vive.

Un graphe ne présente pas de symbole de transition, mais nos principes d'implémentation (cf. Section 6.1.8.2) permettent de le manipuler comme une autre machine. Il accepte notamment la composition, et est donc compact et utilisable.

10.2.6 Processus de sélection et synthèse

10.2.6.1 Processus de sélection

La totalité du processus est gérée par les 4 FSMs construits au cours de l'apprentissage. Aucune pondération n'est nécessaire, puisque les coûts cibles et les coûts de concaténation sont précalculés et mémorisés respectivement dans T_{CM} et dans A_{CO}. Les seuls calculs réalisés par le processus sont :

- Des sommes lorsque deux machines sont composées ensemble.
- La recherche du meilleur chemin dans la dernière phase de la sélection.

L'algorithme du processus de sélection complet est présenté en Pseudocode 28. Il puise les informations linguistiques concernant la phrase dans une structure de données appelée « DLS », pour *Data Layer Structure*. La phase de pré-sélection se déroule dans une itération où chaque cible diphonique est traitée séparément (lignes 2-8). Deux FSAs sont d'abord créés (cf. Figure 10.15). Le premier représente uniquement le diphone en cours (ligne 3), et le second, uniquement les critères linguistiques correspondants (ligne 4). Les deux FSAs sont passés à la méthode `FSM_PRESELECTION` (ligne 5), qui retourne un WFSA contenant l'ensemble des candidats sous la forme de leurs identifiants. Au cours des itérations successives, un WFSA représentant la séquence des candidats pour la totalité de la phrase est créé par concaténation (ligne 6). La concaténation insère inévitablement des transitions ϵ qui sont supprimées lorsque la phrase est complètement construite (ligne 9, et illustration Figure 10.16). La méthode `FSM_SELECTION` choisi enfin la meilleure séquence d'unités dans ce WFSA (ligne 9).

Les algorithmes des méthodes `FSM_PRESELECTION` et `FSM_SELECTION` sont détaillés ci-après.

Pré-sélection. Le Pseudocode 29 présente l'algorithme de pré-sélection. L'algorithme comporte trois étapes. La première étape teste si le diphone appartient aux diphones manquants (ligne 1). Si c'est le cas (ligne 2), le FSA du diphone est remplacé par le FST de substitutions proposées (ligne 4, illustration Figure 10.17), que l'on réduit à un FSA en en projetant la sortie (ligne 5).

La seconde étape construit le FSA cible, « diphone + critères » (ligne 7) et teste si la cible appartient aux cibles manquantes (ligne 8). Si c'est le cas (ligne 9), le FSA cible

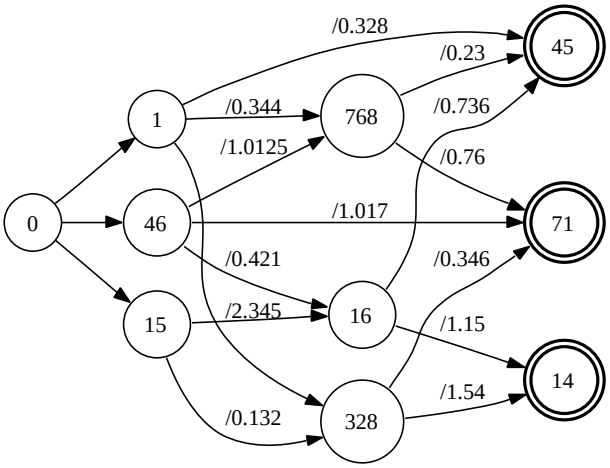
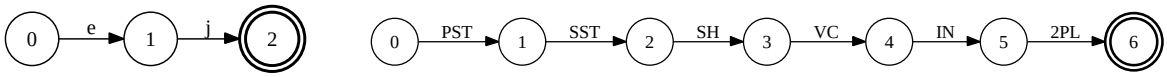


FIG. 10.14: Graphe de concaténation. Les transitions ne sont étiquetées que par le coût de concaténation. Comparée à la taille du WFSA de la Figure 10.10 (b), la taille de celui-ci est donc fortement réduite



(a) Le diphone.

(b) Les critères.

FIG. 10.15: Une cible au format FSA

Require: $pDLS$, une structure de données représentant la phrase en cours,
T_{DM}, T_{CM}, T_{CP} et A_{CO}, calculés à l'entraînement
Ensure: $pBest$, un WFSA contenant la meilleure séquence d'unités sous la forme de leurs identifiants

1: $pSent \leftarrow 0$
2: **for** $i \leftarrow 0$ to $i < (n-1)$ **do**
3: $pDi \leftarrow$ `CREATE_DIPHO_FSM`$(pDLS, i)$
4: $pFeat \leftarrow$ `CREATE_FEATURE_FSM`$(pDLS, i)$
5: $pCand \leftarrow$ `FSM_PRESELECTION`$(pDi, pFeat, T_{DM}, T_{CM}, T_{CP})$
6: $pSent \leftarrow$ `FSM_Concat`$(pSent, pCand)$
7: `FSM_Destroy`$(pCand)$
8: **end for**
9: `FSM_EpsilonFree`$(pSent)$
10: $pBest =$ `FSM_SELECTION`$(pSent, A_{CO})$
11: **return** $pBest$

Pseudocode 28: `FSM_LiONS`

est remplacé par le WFST de substitutions proposées (ligne 11, illustration Figure 10.18), que l'on réduit à un WFSA en en projetant la sortie (ligne 12). A ce stade, il est maintenant certain que les cibles demandées appartiennent aux cibles présentes.

La troisième étape peut donc composer le WFSA avec le FST des cibles présentes (ligne 14), en sachant qu'il y aura toujours un résultat. Pour ne conserver que les identifiants des unités, la sortie du résultat est projetée (ligne 16, illustration Figure 10.19). Cette projection nécessite une suppression des transitions ϵ, puisqu'une cible est décrite par plusieurs transitions, alors qu'une seule transition correspond à un identifiant.

Sélection. Le Pseudocode 30 présente l'algorithme de sélection, fort simple. Le WFSA représentant l'ensemble des candidats pour la phrase est composé avec le WFSA représentant l'ensemble des concaténations autorisées (ligne 1). Dans le résultat, les candidats cibles ont été repondérés par les coûts de concaténation au cours de la composition. Il suffit dès lors de calculer la meilleure séquence du résultat (ligne 2).

Format de sortie. Nous générons toujours un fichier dipho, mais le format de ce fichier a changé (cf. Figure 10.20) :

1. Un diphone est directement référencé par son identifiant, étant donné que l'entraînement, maintenant, crée une table diphonique où un identifiant permet de retrouver toutes les informations concernant le diphone (fichier, début, fin, etc.). Les informations destinées à l'eFriend (demi-phones et durées) sont cependant toujours présentes.
2. Un nouveau type d'unité est généré. Il s'agit de la `PAUSE`, qui est suivie d'une durée exprimée en échantillons. Le principe est qu'en présence d'un diphone terminé par un silence, nous vérifions dans la DLS le type de pause à générer [17]. Cette modification est valable pour l'eFriend également : il réalise une pause de la durée choisie, mais ne s'arrête plus sur les demi-phones « silence » dont les durées sont mises à 0.

10.2.6.2 Synthèse

La synthèse est toujours réalisée à l'aide de l'algorithme Copy-OLA, qui a simplement été modifié de manière à générer l'unité `PAUSE`. Le principe est très simple : au lieu de copier un silence depuis le corpus, un silence artificiel est généré. La suite du processus ne change pas : le silence est superposé et additionné aux unités qui l'entourent, après que les frontières concernées aient été lissées par une fenêtre de Hanning (cf. 10.1.6).

10.2.7 Analyse

Des exemples de synthèse permettant une comparaison des deux versions de LiONS peuvent être écoutés sur notre site [18].

[17] Un fichier d'initialisation, quant à lui, nous renseigne sur la longueur correspondant à la pause en question.

[18] `richardbeaufort.co.nr/phd/2-nuu-LiONS2.0_eval.html`.

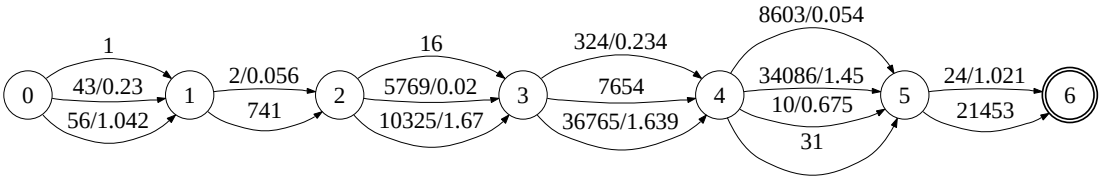


FIG. 10.16: Une phrase au format WFSA. Chaque diphone est représenté par les identifiants de ses unités. Les poids sont les coût cibles

Require: pDi, le FSA du diphone, et $pFeat$, le FSA des critères,
T_{DM}, T_{CM} et T_{CP}, calculés à l'entraînement
Ensure: $pCand$, un WFSA contenant les n meilleurs candidats sous la forme de leurs identifiants

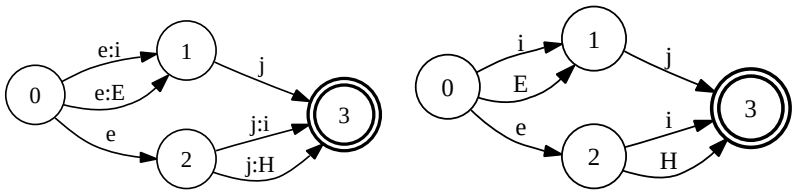
1: $pTmp \leftarrow$ `FSM_Compose`(pDi, T_{DM})
2: **if** $pTmp$ **then**
3: `FSM_Destroy`(pDi)
4: $pDi \leftarrow pTmp$
5: `FSM_Project`$(pDi,$ `OUT`$)$
6: **end if**
7: $pTarget \leftarrow$ `FSM_EpsilonFree`$($`FSM_Concat`$(pDi, pFeat))$
8: $pTmp \leftarrow$ `FSM_Compose`$(pTarget, T_{CM})$
9: **if** $pTmp$ **then**
10: `FSM_Destroy`$(pTarget)$
11: $pTarget \leftarrow pTmp$
12: `FSM_Project`$(pTarget,$ `OUT`$)$
13: **end if**
14: $pCand \leftarrow$ `FSM_Compose`$(pTarget, T_{CP})$
15: `FSM_Destroy`$(pTarget)$
16: `FSM_EpsilonFree`$($`FSM_Project`$(pCand,$ `OUT`$))$
17: **return** $pCand$

Pseudocode 29: `FSM_PRESELECTION`

Require: $pSent$, le WFSA des candidats pour la totalité de la phrase, sous la forme de leurs identifiants
A_{CO}, calculé à l'entraînement
Ensure: $pBest$, un WFSA contenant la meilleure séquence d'identifiants pour la phrase

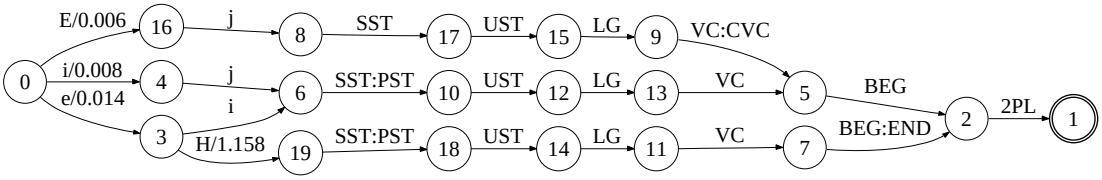
1: $pTmp \leftarrow$ `FSM_Compose`$(pSent, A_{CO})$
2: $pBest \leftarrow$ `FSM_GetBestPath`$(pTmp, 1)$
3: `FSM_Destroy`$(pTmp)$
4: **return** $pBest$

Pseudocode 30: `FSM_SELECTION`

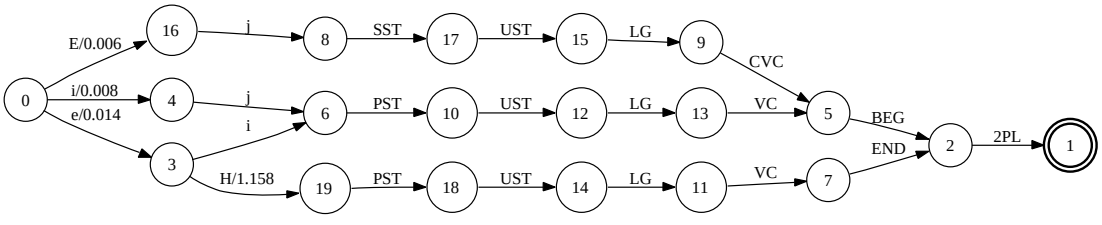


(a) résultat initial (b) après projection

FIG. 10.17: Substitution d'un diphone manquant

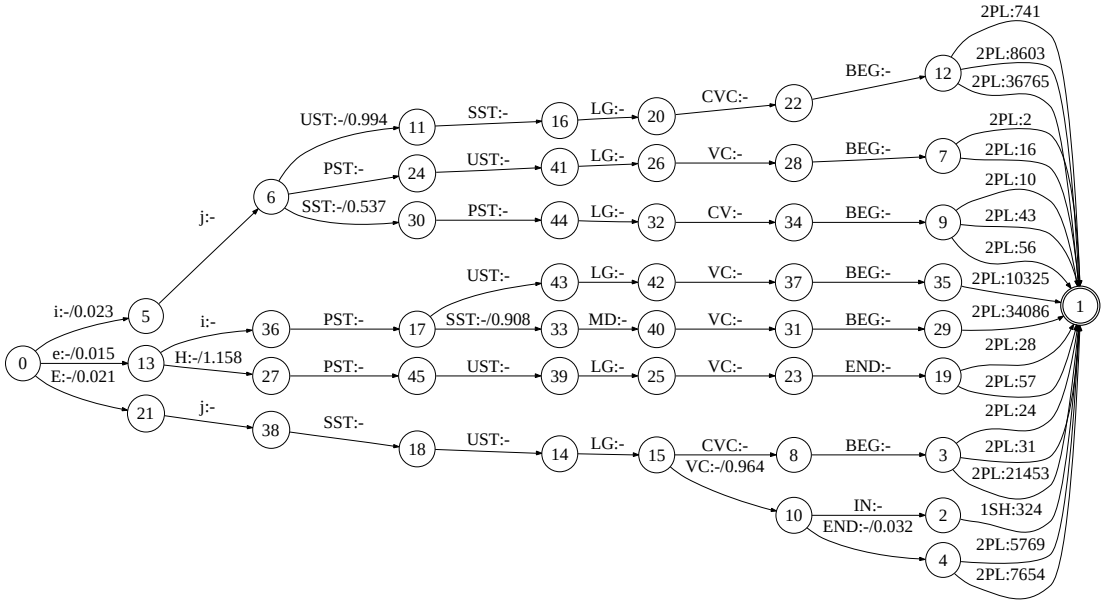


(a) résultat initial

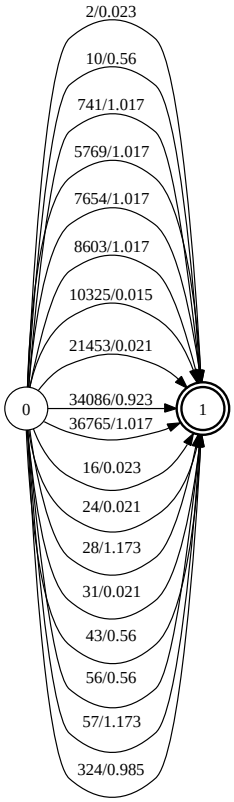


(b) après projection

FIG. 10.18: Substitution d'une cible manquante



(a) résultat initial



(b) après projection et suppression-ϵ

FIG. 10.19: Des candidats aux identifiants

Modélisation de la base. Pour une phrase de la base, la sélection ne retourne plus toujours la phrase elle-même. Ceci n'indique cependant ni une désynchronisation de l'étiquetage entre la base et l'algorithme, ni une erreur de l'algorithme de sélection. Ce résultat est simplement dû à la nouvelle gestion des pauses, puisque, dans la machine de concaténation :

- Toute unité terminée par un silence et qui ne se situe pas en fin de phrase peut être suivie par toute unité qui commence par un silence et qui ne se situe pas en début de phrase.
- Ces transitions ne sont pas pondérées.
- Aucune préférence n'est accordée aux unités contiguës.

Or, certaines phrases de notre corpus contiennent des séquences de mots identiques *entre pauses*. En présence de ces phrases, l'algorithme n'a aucune information susceptible de distinguer les séquences en question, et choisit simplement celle qui arrive *par hasard* en tête du classement.

Qualité de la parole. Si l'analyse objective de la qualité d'une parole de synthèse est difficile, l'identification des causes d'une dégradation ou d'une amélioration est certainement délicate : comme nous l'avons mentionné précédemment (cf. Section 8.5.3), de nombreux paramètres, intrinsèques et extrinsèques au module de sélection, interviennent dans le processus de sélection. L'analyse que nous proposons ici est donc certainement sujette à discussion et n'a aucune valeur de vérité.

Dans l'ensemble, cette seconde version de LiONS propose une qualité de synthèse fort comparable à celle de la première version, ce qui est rassurant, puisque les bases du système et la méthode globale n'ont pas changé. On constate en outre que les quelques problèmes qui avaient été relevés dans LiONS 1 sont résolus ou fortement atténués dans LiONS 2 :

1. Les problèmes de durée des phonèmes sont beaucoup moins fréquents. Ceci est particulièrement marquant à l'endroit des pauses, où les allongements sont nettement mieux gérés. On peut donc considérer que le critère linguistique de distance à la pause et le critère acoustique de durée moyenne contribuent ensemble à discerner les unités adéquates.
2. La courbe mélodique descend maintenant systématiquement en fin de phrase, sauf lorsqu'aucune unité de fin de phrase n'a pu être trouvée dans le corpus. Les erreurs sont ici toutes attribuables à la couverture du corpus.
3. Les pauses sont toujours réalisées correctement, puisqu'elles sont générées par le synthétiseur en fonction des indications de la sélection, et non plus simplement sélectionnées dans le corpus.
4. La courbe prosodique globale de la phrase est mieux gérée : l'intonation ne varie plus de manière inopinée. Ceci semble confirmer que l'équilibre recherché dans la répartition des transitions de l'automate de concaténation apporte une amélioration.

```
; Bonjour, je cherche un restaurant.
39164   _     0     b   790
39165   b     733   o   1145
39166   o     1139  Z   1121
39167   Z     1125  u   1016
39168   u     1009  R   1056
39169   R     1175  _   0
PAUSE   8000
22801   _     0     Z   520
22802   Z     547   @   349
34975   @     619   S   1095
23316   S     1055  E   508
23317   E     552   R   270
35277   R     468   S   882
35278   S     809   9   898
23846   9     783   R   376
32910   R     615   E   580
32911   E     578   s   803
32912   s     744   t   756
9246    t     808   o   615
9247    o     565   R   397
10089   R     341   a   1434
11885   a     1099  _   0
PAUSE   9600
```

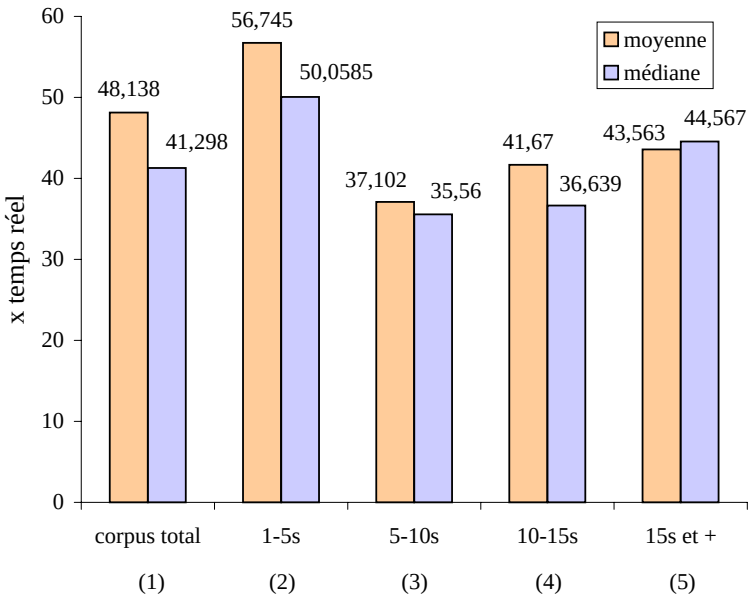
FIG. 10.20: Fichier dipho

Cette qualité générale, doublée de ces améliorations significatives, tend à prouver que :

1. Les critères linguistiques du coût cible sont pertinents et suffisants : ils apportent l'information nécessaire à la gestion de la courbe prosodique et de la durée.
2. Les critères acoustiques du coût de concaténation sont pertinents et suffisants : il y a un équilibre entre les critères qui gèrent la fluidité et ceux qui gèrent la durée.
3. Les contraintes ajoutées dans l'automate de concaténation (distribution des transitions, gestion des unités acceptées à la pause) apportent une réelle amélioration dans la gestion prosodique.

Temps de sélection. Nous avons réalisé des tests sur 950 phrases de longueur variable. La Table 10.10 récapitule les temps de sélection obtenus et montre que le processus, qui était $\frac{1}{4}$ temps réel dans la première version de LiONS, est devenu en moyenne 48 fois temps réel. Les moins bons résultats ont été observés sur les phrases de 5 à 10s, alors qu'ils s'améliorent sur les phrases plus longues. La médiane est presque systématiquement inférieure à la moyenne, et le minimum est à 35 fois temps réel.

Les systèmes de l'état de l'art qui indiquent leur temps de sélection sont au mieux temps réel, sauf Allauzen *et al.* (2004), qui est 2,6 fois plus rapide que le système temps réel de Beutnagel *et al.* (1999a). Notre système propose donc un temps de sélection nettement inférieur au meilleur des systèmes précédents.



TAB. 10.10: Temps de sélection. Moyennes et médianes pour le corpus entier (1) et par intervalles de durée (2-5)

Place requise. La place disque nécessaire pour stocker nos FSMs au format binaire est approximativement de 82,8 Mo [19]. Ce n'est certainement pas rien. Il est malheureusement impossible de comparer notre système aux systèmes de l'état de l'art sur ce point, qui ne fournissent aucune information à ce sujet.

Le FSM le plus gourmand est évidemment le WFSA des coûts de concaténation, qui prend à lui seul 79,7 Mo. Cependant, l'ensemble des données acoustiques prélevées sur le corpus et nécessaires au calcul des coûts représente initialement 323,1 Mo. Il serait donc plus coûteux de ne pas précalculer les coûts, puisque ces données devraient être conservées pour permettre le calcul des poids en cours de processus.

Notons cependant qu'une alternative existe. Comme certains systèmes de l'état de l'art (Black & Campbell 1995, Bulyko & Ostendorf 2001), nous aurions pu rassembler les réalisations acoustiques en partitions relativement « similaires » par quantification vectorielle (Linde *et al.* 1980, Gray 1984). Dans ce cas, la distance entre deux réalisations acoustiques est déterminée par rapport aux partitions auxquelles les réalisations appartiennent. Ceci permet évidemment de réduire considérablement la place requise, mais inévitablement au détriment de la qualité. La place requise par le FSM n'est donc pas dû au mode de représentation choisi, mais à notre volonté d'estimer la distance entre réalisations directement à partir de celles-ci.

Le point le plus important est certainement que notre mode de représentation binaire prend exactement la même place en mémoire que sur le disque (cf. Section 6.1.5). Une mémoire vive de 128 Mo est ainsi suffisante pour ce processus, qui devrait donc pouvoir être exécuté sur les derniers modèles d'agenda électronique, pourvus de 256 Mo de mémoire [20].

[19] 13,4 Ko + 1,8 Mo + 1,3 Mo + 79,7 Mo.

[20] Nous donnons un exemple d'utilisation de notre bibliothèque sur plateforme embarquée en correction orthographique (cf. Partie III).

Chapitre 11

Conclusion

Cette partie a été entièrement consacrée au processus de synthèse par sélection d'unités non uniformes, dernier état de l'art en synthèse de la parole à partir du texte.

11.1 Principe

Afin d'obtenir une parole aussi naturelle que possible, ce type de synthèse est basé sur l'utilisation d'un corpus d'unités de parole contenant plusieurs réalisations acoustiques de chaque unité, afin d'une part de faciliter la gestion des phases de coarticulation et d'autre part de limiter le traitement du signal autant que faire se peut. Le système comprend de ce fait une étape de sélection d'unités basée sur la résolution d'un double coût « cible–concaténation ».

11.2 Etat de l'art

Notre présentation de l'état de l'art dans le domaine a mis au jour les limites du modèle – ou en tout cas celles des systèmes proposés :

1. Le choix des critères de sélection est délicat. L'état de l'art a oscillé entre informations acoustiques et linguistiques, mais sans trouver le juste équilibre entre les deux et sans proposer les critères adéquats.
2. La pondération de ces critères est difficile, et l'état de l'art n'a pu proposer de pondération automatique convaincante.
3. La méthode d'optimisation ne peut modifier involontairement la méthode de sélection. Or, les optimisations efficaces proposées dans l'état de l'art ont soit modifié involontairement l'algorithme de sélection, soit rendu impossible une quelconque modification des paramètres du système.

11.3 Méthode proposée

La méthode de sélection que nous avons proposée tâche de dépasser les limites relevées dans l'état de l'art. Elle a été élaborée en deux temps, ou deux versions successives.

11.3.1 Première version

Objectifs. La première version de cette méthode se voulait principalement une preuve de concept. Elle a voulu :

1. Evaluer la pertinence de n'utiliser que des critères linguistiques au niveau de la pré-sélection, laissant l'évaluation de la concaténation à des critères acoustiques.
2. Estimer la possibilité de pondérer automatiquement les critères de la pré-sélection sur la base de leur gain d'information.

Cette première version n'a par contre pas cherché à optimiser les ressources utilisées, ni le temps de traitement nécessaire.

Evaluation. Les constats suivants ont été faits :

1. Les notes attribuées par les utilisateurs aux phrases du corpus sont intéressantes. Elles montrent le manque de qualité – peut-être subjective – de la voix originale et nous laissent supposer que l'estimation de la qualité de la parole de synthèse s'en ressent.
2. De manière générale, l'évaluation de la parole de synthèse par les utilisateurs est plutôt positive. La prosodie, la concaténation et le confort d'écoute sont ressentis comme normaux, tandis que l'intelligibilité est fortement mise en avant. Aucun critère ne reçoit une mauvaise note globale.
3. Dans l'ensemble, la synthèse produite par LiONS est de très bonne qualité. Malgré le manque de couverture du corpus, elle présente très peu de problèmes de concaténation et propose une prosodie tout à fait acceptable. Le coût cible linguistique, le coût de concaténation acoustique et la pondération automatique des critères du coût cible participent donc à un système équilibré.
4. Une écoute attentive de la parole générée par cette première version révèle cependant quelques caractéristiques gênantes :
   (a) Un phonème est parfois anormalement long ou court. Or, de nombreux critères suprasegmentaux gèrent la prosodie.
   (b) Il arrive que la courbe mélodique de fin de phrase ne descende pas, ce qui laisse la phrase en suspens, comme inachevée.
   (c) Une pause donnée n'est pas réalisée par un silence de longueur constante : pour une même pause, la durée du silence varie parfois fortement. Lorsqu'il y a erreur, le silence est souvent trop long, de sorte qu'une virgule, par exemple, peut être ressentie comme un point.

(d) La courbe prosodique globale de la phrase varie parfois de manière inopinée, montant ou descendant excessivement.

5. Le système n'est pas temps réel, mais $\frac{1}{4}$ temps réel. Cependant, comme nous l'avons déjà signalé, l'optimisation ne faisait pas partie des objectifs de ce premier système.

11.3.2 Seconde version

Objectifs. Le seconde version de cette méthode a été développée avec le souci de résoudre les problèmes relevés dans la première version, et d'optimiser l'algorithme initial tout en le respectant :

1. Les critères de présélection finalement retenus dans cette seconde version sont le résultat d'une réflexion linguistique, qui a tâché de dégager les critères suprasegmentaux discriminants, capables de diriger la courbe prosodique sans la déterminer explicitement. Cette réflexion linguistique a conduit à deux hypothèses : la première est l'intérêt de choisir le diphone comme unité de sélection, du fait de l'ensemble d'informations qu'il condense naturellement, et la seconde est l'importance d'ajouter un critère de distance de l'unité par rapport à la pause, au vu de l'influence de la pause sur la longueur des phonèmes. Cette réflexion linguistique a ainsi permis de passer de 36 critères à 6.
2. Un nouveau critère a été ajouté au coût de concaténation : la durée moyenne du phonème, dont l'objectif est d'éviter les phonèmes anormalement longs ou courts.
3. L'optimisation du processus a été réalisée en divisant les phases de sélection entre 4 machines à états finis qui respectent l'algorithme initial : (1) vérification de l'existence du diphone, (2) recherche de candidats proches pour une cible donnée, (3) conversion de ces candidats en identifiants, (4) recherche du meilleur chemin dans le treillis d'identifiants. Toutes les valeurs sont précalculées dans les machines, de sorte que seules quelques sommes et la recherche du meilleur chemin restent à calculer en cours de traitement.
4. Afin d'améliorer l'algorithme initial, de nouvelles contraintes ont été directement exprimées dans les machines à états finis. C'est le cas, par exemple, de la réduction du nombre d'unités finales acceptées. Ce type de modifications n'est en rien une obligation du fait de la structure de données utilisée, mais une facilité offerte par celle-ci.
5. Les pauses sont générées et non plus sélectionnées dans le corpus, de manière à assurer la longueur de la pause en fonction de son type.

Evaluation. Dans l'ensemble, cette seconde version de LiONS propose une qualité de synthèse fort comparable à celle de la première version. Nous avons cependant constaté que les quelques problèmes qui avaient été relevés dans la première version sont résolus ou fortement atténués dans cette seconde version :

1. Les problèmes de durée des phonèmes sont beaucoup moins fréquents. Ceci est particulièrement marquant à l'endroit des pauses, où les allongements sont nettement mieux gérés. On peut donc considérer que le critère linguistique de distance à la pause et le critère acoustique de durée moyenne contribuent ensemble à discerner les unités adéquates.
2. La courbe mélodique descend maintenant systématiquement en fin de phrase, sauf lorsqu'aucune unité de fin de phrase n'a pu être trouvée dans le corpus. Les erreurs sont ici toutes attribuables à la couverture du corpus.
3. Les pauses sont toujours réalisées correctement, puisqu'elles sont générées par le synthétiseur en fonction des indications de la sélection, et non plus simplement sélectionnées dans le corpus.
4. La courbe prosodique globale de la phrase est mieux gérée : l'intonation ne varie plus de manière inopinée. Ceci semble confirmer que l'équilibre recherché dans la répartition des transitions de l'automate de concaténation apporte une amélioration.
5. Le processus, qui était $\frac{1}{4}$ temps réel dans la première version, est devenu en moyenne 48 fois temps réel. Les moins bons résultats ont été observés sur les phrases de 5 à 10s, alors qu'ils s'améliorent sur les phrases plus longues. La médiane est presque systématiquement inférieure à la moyenne, et le minimum est à 35 fois temps réel.
6. La place disque nécessaire pour stocker nos FSMs au format binaire atteint les 82,8 Mo, dont 79,7 Mo pour la machine des concaténations. La place nécessaire est donc conséquente, mais il faut noter que :
   (a) L'ensemble des données acoustiques correspondant à la machine de concaténation dépasse les 300 Mo.
   (b) La taille de la machine de concaténation est réduite de moitié grâce à la compaction de l'automate sous la forme d'un graphe.
   (c) La place mémoire requise pour charger et utiliser les machines binaires est identique à la place disque utilisée, ce qui est relativement réduit au vu des mémoires actuellement disponibles.

11.4 Bien-fondé du modèle

Le modèle que nous avons proposé a été guidé par les postulats suivants :

Postulat 10.1.1 (Unité de base). *La seule unité qui autorise un traitement du signal limité au point de concaténation est le diphone, étant donné que dans tous les cas, la concaténation est réalisée en partie stable du signal. L'apparente impossibilité de couvrir le diphone à l'aide d'un corpus de parole est simplement le reflet de la richesse réelle de la langue, que des unités non contextuelles comme le phone ou le demi-phone ne modélisent pas correctement.*

Postulat 10.1.2 (Accent de mot). *En français, l'accent de mot est conservé, mais s'atténue par rapport à l'accent de groupe. Deux* degrés *d'accentuation existent donc en français : un degré* majeur *au niveau du groupe et un degré* mineur *au niveau du mot.*

Postulat 10.2.1 (Contraintes de concaténation). *La structure de représentation ne peut modifier l'algorithme de sélection de manière involontaire. Elle peut par contre être l'occasion d'ajouter des contraintes supplémentaires qui améliorent l'algorithme.*

L'analyse des résultats de synthèse obtenus avec les deux versions de notre méthode démontre, quant à elle, le bien-fondé de nos hypothèses :

11.4.1 Sur les principes de sélection

Hypothèse 10.1.1 (Critères de sélection). *Un système de synthèse par sélection peut produire de la parole de haute qualité en n'utilisant* que *des informations linguistiques segmentales et suprasegmentales au niveau du coût cible, pour autant que le coût de concaténation soit* exclusivement *dirigé par des critères acoustiques.*

Hypothèse 10.1.2 (Pondération). *Les critères linguistiques du coût cible dirigent une prise de décision prosodique. Ils sont donc de nature à être départagés par entropie.*

Hypothèse 10.1.3 (Pondération par phonème). *Du fait de leurs différences articulatoires, les phonèmes se comportent différemment les uns des autres dans un même contexte d'élocution. De ce fait, une seule pondération des caractéristiques linguistiques ne serait pas pertinente. Il est préférable de pondérer les caractéristiques pour chaque phonème indépendamment.*

Hypothèse 10.2.2 (Pré-sélection par diphone). *Le diphone est une unité qui condense suffisamment d'informations segmentales pour autoriser la suppression de toutes les informations segmentales de la pré-sélection. La structure du diphone est en outre l'occasion d'exprimer de manière concise de nombreux critères suprasegmentaux.*

Hypothèse 10.2.3 (Durée et pause). *La durée syllabique est avant tout une conséquence du caractère éminemment mécanique de la respiration. Parmi les nombreuses caractéristiques de la respiration, la pause est particulièrement discriminante. En français, la syllabe qui précède directement une pause est toujours allongée, l'importance de son allongement dépendant de la structure interne de la syllabe.*

11.4.2 Sur les principes d'optimisation

Hypothèse 10.2.1 (Sélection par FSMs). *Les machines à états finis constituent un mode de représentation idéal pour optimiser le processus de sélection d'unités non uniformes. Ce*

mode de représentation permet de précalculer la pré-sélection, le coût cible et le coût de concaténation tout en respectant l'algorithme de sélection initial.

Hypothèse 10.2.4 (Contraintes de concaténation). *Une machine à états finis pondérée permet de respecter l'algorithme de sélection, tout en refusant certaines transitions nuisibles à la qualité globale de la sélection.*

Ces hypothèses s'inscrivent par ailleurs dans la lignée de notre hypothèse initiale :

Hypothèse 1 (Diviser pour mieux régner). *Pour autant qu'une tâche complexe puisse être analysée comme une succession d'étapes simples, les machines à états finis constituent l'outil idéal pour représenter cette succession d'étapes simples de manière aisée.*

11.4.3 Sur le caractère multilingue

Le système de sélection est totalement indépendant du contenu des machines, qui encapsulent la totalité des informations propres à la langue. De ce fait, les caractéristiques retenues pourraient varier d'une langue à l'autre, sans que l'algorithme en soit affecté. Ceci confirme donc notre hypothèse initiale :

Hypothèse 2 (Externalisation des données). *Pour autant que les langues traitées partagent suffisamment de similarités, les machines à états finis autorisent l'externalisation de l'ensemble des traitements dépendant de la langue. Tout l'art réside dès lors dans la détection, au sein d'un processus, des traitements qui ressortissent à la langue.*

11.5 Les apports des machines à états finis

Les machines à états finis ont trouvé, dans la sélection d'unités non uniformes, un domaine d'application mettant en évidence leurs qualités indéniables. Elles constituent un cadre pertinent, ajustable, simple et élégant pour optimiser le processus de sélection.

Pertinent. Cet outil permet d'optimiser la sélection sans modifier l'algorithme initial. Il se distingue en ceci des arbres de décision ou des modèles de langue proposés dans l'état de l'art.

Ajustable. Les paramètres du système sont indépendants du mode de représentation et peuvent être réglés de manière fine. Ceci n'est pas le cas d'un modèle de langue, dont les paramètres ne peuvent plus être gérés séparément.

Simple. Le processus se divise en quelques étapes élémentaires, résultats de l'application du principe « diviser pour mieux régner ». De ce fait, les machines précalculées ne contiennent que les données nécessaires, et les résultats d'une étape du processus peuvent être réduits au strict minimum avant d'être utilisés par l'étape suivante. Cette réduction réalisée est une opération supplémentaire, mais elle permet globalement d'accélérer le processus, puisqu'elle contient la recherche dans un espace relativement limité.

Elégant. Les améliorations que nous avons apportées à l'algorithme original ne sont pas imposées, mais plutôt facilitées ou autorisées par les FSMs. Les machines à états finis ont l'élégance d'exprimer les contraintes sur les données plutôt que dans l'algorithme lui-même.

11.6 Une synthèse cependant aléatoire...

La synthèse par sélection d'unités non uniformes génère de la parole de haute qualité et très intelligible, que l'on peut considérer à juste titre comme proche de la parole humaine. L'apparition de cette synthèse a d'ailleurs stimulé l'utilisation de la synthèse de la parole dans de nombreuses applications interactives. C'est le cas, par exemple, du guidage vocal des systèmes de navigation.

Ces qualités ne doivent cependant pas faire oublier que les résultats obtenus sont encore aléatoires : il est impossible d'assurer la qualité des concaténations réalisées et la justesse de la courbe prosodique générale, parce qu'il est impossible de prévoir quelles unités seront sélectionnées par le système. Les résultats obtenus sont en outre fort dépendants du savoir-faire mis en œuvre.

Cet état de l'art n'est donc qu'une étape, certainement pertinente et efficace, vers la modélisation d'un système de synthèse dont la parole serait à la fois naturelle et ... *constante.*

Troisième partie

Correction orthographique

Chapitre 12

Introduction

12.1 De l'importance de l'analyse linguistique en synthèse

En synthèse de la parole à partir du texte, le module de traitement du signal génère la parole à partir d'une représentation non ambiguë du texte, construite par un module de traitement automatique de la langue (TAL).

Afin de guider le module de traitement du signal, la représentation non ambiguë doit au moins être constituée de la séquence de phonèmes correspondant au texte et d'un ensemble d'informations prosodiques. Or, phonèmes et prosodie ne peuvent être déduits du texte brut :

1. *Séquence de phonèmes.* La séquence de phonèmes correspondant à un texte peut être construite en deux étapes. La première étape consiste à phonétiser les mots pris isolément. La seconde étape consiste à traiter les phénomènes phonétiques qui se produisent en frontière de mot, dans le contexte de la phrase. Dans les deux cas, une information est nécessaire au processus : la catégorie syntaxique de chaque mot de la phrase. Les homographes hétérophones, par exemple, se prononcent différemment selon la catégorie qui leur est attribuée (*couvent*, *président*, etc.), tandis que certaines liaisons dépendent des catégories syntaxiques en contact dans la phrase (*les oiseaux*).
2. *Informations prosodiques.* Nous avons vu que, selon les systèmes (cf. Chapitre 9), ces informations prosodiques sont de type acoustique ou de type linguistique. Dans les deux cas, les catégories syntaxiques des mots de la phrase sont également nécessaires. Dans notre système par exemple (cf. Section 10.2.2), les groupes rythmiques correspondent à des séquences de catégories, et la syllabation de la séquence phonétique tient compte des groupes rythmiques pour poser certaines frontières syllabiques.

La génération de la représentation non ambiguë ne peut donc être réalisée qu'après une analyse linguistique du texte, qui doit attribuer à chaque mot de la phrase sa catégorie syntaxique. Ce résultat est généralement atteint en trois temps :

1. Un pré-processeur découpe d'abord le texte en phrases et en mots. Deux remarques sont à faire à ce sujet. Premièrement, les frontières entre phrases sont parfois floues, du fait de l'ambiguïté entre ponctuations et délimiteurs d'abréviations. Deuxièmement, le *mot* est une notion vague et discutée, sur laquelle nous ne nous étendons pas dans le cadre de cette introduction. Disons que le mot est considéré dans un système classique comme « toute suite de caractères alphabétiques délimitée par des espaces ou des signes de ponctuation ».
2. Un analyseur morphologique attribue ensuite à chaque mot l'ensemble des catégories syntaxiques qui lui correspondent. De nombreux mots possèdent plusieurs analyses. Parmi ceux-ci, on trouve les homographes hétérophones que nous avons déjà cités, mais également des homophones comme « ferme » (Nom, Adj. ou Verbe) ou « donne » (Nom ou Verbe).
3. Un analyseur syntaxique détermine enfin la bonne catégorie pour chaque mot en fonction du contexte de la phrase. Nous attirons ici l'attention sur le fait que l'analyse syntaxique en synthèse n'est en rien élaborée : elle ne construit pas la structure syntaxique de la phrase, parce que celle-ci n'est pas utilisée par les modules suivants.

C'est de la nécessité de ces différents niveaux d'analyse que se dégage l'architecture générale d'un module TAL en synthèse (cf. Figure 12.1), architecture qui illustre le fait que l'analyse linguistique constitue la pierre d'angle du système de synthèse : sans une analyse linguistique robuste, pas de parole de synthèse de qualité...

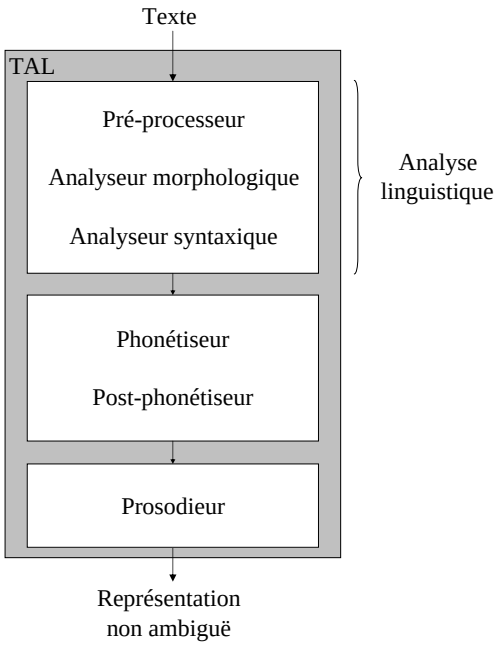


FIG. 12.1: Architecture générale d'un module TAL

12.2 Positionnement du problème

Sans entrer à ce stade dans la description d'une typologie complète des erreurs que peut contenir un texte, disons que les fautes d'orthographe peuvent résulter en des non-mots ou en des mots connus, mais incorrects dans le contexte de la phrase.

Or, étant donné que la synthèse *prononce* le texte, les fautes que celui-ci contient risquent d'être audibles :

1. Certaines fautes sont audibles indépendamment du contexte. Ce sont les non-mots (*élève* → *élèbe*, *élèbve*, *élèe*).
2. D'autres fautes ne sont audibles que dans certains contextes. Ce sont par exemple certaines liaisons manquantes (*les beau_ enfants* >< *les belle_ filles*).
3. Dans tous les cas, les fautes d'orthographe mettent en péril l'analyse de la phrase complète, parce que le système ne possède pas les bonnes informations. Or, si l'analyse linguistique échoue, certains termes – au demeurant corrects – seront mal analysés. L'analyse linguistique risque dans ce cas d'induire en erreur les modules de phonétisation et de gestion de la prosodie :
   (a) Les homographes hétérophones mal analysés seront mal prononcés. Imaginons, par exemple, que *couvent* soit analysé comme un nom alors qu'il s'agit d'un verbe...
   (b) Les phénomènes de post-phonétisation pourraient ne pas être traités correctement. Des liaisons pourraient par exemple ne pas être réalisées ou être réalisées à tort.
   (c) L'analyse prosodique peut être faussée. Dans notre système par exemple, les groupes rythmiques pourraient être erronés, ce qui aura une influence sur la qualité de la syllabation, mais également sur le choix des unités de parole.

Ces quelques exemples mettent en évidence le nombre élevé de conséquences audibles que de « simples » fautes d'orthographe peuvent entraîner en synthèse de la parole, alors que l'être humain, confronté aux mêmes fautes, pourra très souvent en faire abstraction et lire normalement le texte qui les contient.

Il est donc primordial, pour la robustesse du système de synthèse dans son ensemble et la qualité de la parole générée en particulier, qu'un système performant de correction orthographique intègre le module d'analyse linguistique.

Les premiers travaux en correction orthographique ont vu le jour dans les années '60. Pourtant, la correction des mots d'un texte constitue encore aujourd'hui un domaine important de la recherche scientifique. En effet, bien que d'excellents systèmes de correction aient été mis au point, de nombreuses applications sont de nos jours encore vulnérables aux erreurs présentes dans les textes.

C'est le cas, par exemple, de la reconnaissance optique de caractères, qui ne fonctionne correctement que sur du texte propre, présenté dans des polices de caractères standards (Burr 1987, Goshtasby & Ehrich 1988, Ho *et al.* 1991, Jones *et al.* 1991). C'est le cas également, comme nous venons de le montrer, de la synthèse de la parole, qui doit rendre audible tout texte qui lui est présenté, et nécessite dès lors une détection et une correction des erreurs de très bonne qualité (Kukich 1990, Tsao 1990, Kernighan 1991).

La recherche dans le domaine de la correction orthographique n'est donc pas arrivée à son terme.

12.3 Plan de la partie

L'objectif de cette partie est principalement de présenter les réflexions et les développements relatifs à l'intégration d'un système de correction orthographique dans l'analyse linguistique de notre système de synthèse, eLite. Dans une moindre mesure, nous présentons également un système beaucoup moins ambitieux, mais efficace, qui s'inscrit dans la cadre de la reconnaissance de caractères. Nous avons réparti les différentes matières, nécessaires à la compréhension du travail réalisé, entre 5 chapitres :

Chapitre 13 : présentation des principes de l'analyse linguistique d'eLite. Ce chapitre dresse dans le même temps un état de l'art succinct de l'analyse morpho-syntaxique.

Chapitre 14 : état de l'art de la correction orthographique, tous domaines confondus.

Chapitre 15 : cœur-même de cette partie, ce chapitre décrit la correction orthographique que nous proposons en synthèse. Notre modèle se veut global, dans le sens où il tâche de reporter au plus loin le choix des corrections. Il a nécessité la refonte complète de l'analyse morpho-syntaxique d'eLite.

Chapitre 16 : présentation de la correction en reconnaissance de caractères. L'application concernée est dédiée aux scènes naturelles, et est exécutée sur plateforme embarquée.

Chapitre 17 : conclusion de cette partie, dans laquelle nous faisons le point sur la portée du travail qui a été réalisé, et l'intérêt des modèles proposés.

Chapitre 13

Présentation de l'analyse linguistique

eLite [1] signifie « *Enhanced, Linguistically-based Text-to-speech synthesis system* ». Commencé en septembre 2001 et conçu initialement afin de fournir aux chercheurs du Centre de recherche une plateforme de test pour de nouveaux algorithmes utiles à la synthèse, ce système, qui a été décrit globalement dans (Beaufort & Ruelle 2006), est rapidement devenu un logiciel stable, robuste et rapide, dont des versions de démonstration sont disponibles sur le site du groupe [2].

L'analyse linguistique d'eLite est le résultat d'une réflexion de groupe. L'objectif, lors de sa conception, a été de tirer le meilleur parti de l'état de l'art dans les différents domaines concernés : pré-traitement, analyse morphologique et analyse syntaxique. Bien sûr, les choix réalisés restent contestables. Ils répondent cependant tous au même souci d'*efficacité* : en synthèse, l'analyse linguistique peut rester relativement superficielle, pour autant qu'elle fournisse aux modules suivants (phonétisation, prosodie, synthèse) les informations dont ils ont besoin.

eLite n'est cependant pas un simple condensé des algorithmes standard de la littérature. La conception de ce système de synthèse a été l'occasion de proposer des concepts nouveaux et quelques approches originales. En ce sens, eLite représente certainement un état de l'art avancé de l'analyse linguistique en synthèse.

En présentant l'analyse linguistique d'eLite, ce chapitre constitue donc également un rapide survol de l'état de l'art en analyse morphologique et en analyse syntaxique.

En Section 13.1, nous commençons par fixer les valeurs des quelques notions morphologiques que nous manipulons dans ce chapitre et les suivants. La Section 13.2 présente ensuite les concepts et la structure de données qui font la particularité d'eLite, et fait le point sur notre rôle dans l'implémentation du système.

Sur cette base, les Sections 13.3, 13.4 et 13.5 décrivent séparément les trois modules de l'analyse linguistique. Nous terminons en Section 13.6 par un point sur les avantages et

[1] eLite se prononce [ɪlaɪt].
[2] www.multitel.be/TTS.

les inconvénients de l'analyse décrite.

13.1 Définitions

Les concepts de la morphologie varient ou changent de valeur selon les théories linguistiques. Ce débat n'étant pas l'objectif de ce document, les pages qui suivent se limiteront aux quelques notions de cette section, dans l'acception que nous leur donnons.

Définition 13.1.1 (Morphologie)**.** *La morphologie est la branche de la linguistique qui étudie les morphèmes de la langue et leurs modes de combinaison pour constituer les formes lexicales de la langue.*

Définition 13.1.2 (Morphème)**.** *Le morphème est l'unité minimale de signification. Le morphème est une unité abstraite, une image acoustique mentale, résultat de la combinaison d'une ou de plusieurs unités distinctives, les phonèmes (cf. Définition 8.3.2). Le morphème est le matériau de construction des formes d'une langue.*

Définition 13.1.3 (Lexème, radical)**.** *Un* lexème *ou* radical *est un morphème considéré comme la plus petite unité lexicale, base de la création de formes autonomes de la langue. Le lexème, dans certains cas, constitue une forme autonome. Il s'agit alors d'un lexème libre. Dans le cas contraire, il s'agit d'un lexème lié.*

Définition 13.1.4 (Affixe, préfixe, suffixe)**.** *Un affixe est un morphème qui ne peut exister seul, mais se combine à un radical pour construire une forme autonome de la langue. Parmi les affixes, on distingue principalement les préfixes, qui précèdent le radical dans la forme, et les suffixes, qui suivent le radical dans la forme.*

Définition 13.1.5 (Forme lexicale)**.** *Une forme lexicale est une forme autonome de la langue, constituée soit d'un radical libre, soit d'un radical et d'un ou de plusieurs affixes soudés dans la graphie.*

Définition 13.1.6 (Mot)**.** *Le mot est une forme lexicale.*

Cette définition courte fixe pour ce document la signification du *mot*, dont le débat en linguistique n'est certainement pas terminé. Le mot, selon cette définition, contient un et un seul radical. Graphiquement, il ne contient donc pas d'espace ni d'apostrophe, étant donné que ces caractères relient toujours plusieurs radicaux. Le mot peut par contre contenir le trait d'union, qui unit parfois un affixe à une forme constituée d'un seul radical.

De la sorte, nous considérons que *pomme de terre* et *aujourd'hui* ne sont pas des mots, mais des mots composés, alors que *pré-évaluation* est un mot.

Définition 13.1.7 (Trait grammatical)**.** *Le trait grammatical est une caractéristique permettant de décrire en détail les formes lexicales d'une langue d'un point de vue grammatical. On distingue les traits intrinsèques au mot, indépendants du contexte, et les traits dépendant du contexte de l'énoncé. Les valeurs d'un trait grammatical dépendent de la langue.*

Parmi les traits grammaticaux intrinsèques, on compte entre autres :

- La nature : nom, verbe, adjectif, etc.
- Le genre : masculin, féminin, neutre.
- Le nombre : singulier, pluriel, neutre.
- Le mode : indicatif, subjonctif, etc.
- Le temps : présent, imparfait, etc.
- La personne : 1, 2, 3.

Le trait dépendant du contexte qui est le plus fréquent est la fonction grammaticale : sujet, complément d'objet direct, etc.

Définition 13.1.8 (Désinence)**.** *Une désinence est une description analytique d'un suffixe grammatical au moyen d'un ou de plusieurs traits grammaticaux.*

En français par exemple, la désinence nominale combine les traits « genre, nombre », et la désinence verbale, « voix, mode, temps, (aspect,) personne, nombre ».

Définition 13.1.9 (Flexion)**.** *La flexion est l'application d'une désinence à un radical pour dénoter les traits grammaticaux voulus.*

Une langue qui recourt à la flexion est dite *flexionnelle*. Il arrive que la flexion ne soit pas identifiable dans la forme lexicale. En français, c'est généralement le cas du « masculin, singulier », raison pour laquelle cette flexion est considérée comme neutre. A titre d'exemple, on comparera *bon* « masculin, singulier » à *bonnes* « féminin, pluriel ».

Définition 13.1.10 (Paradigme)**.** *Le paradigme est l'ensemble des formes lexicales obtenues à partir d'un radical en lui appliquant l'ensemble des flexions possibles d'une désinence donnée.*

Définition 13.1.11 (Lemme)**.** *Le lemme est la forme lexicale d'un paradigme considérée comme canonique et choisie pour constituer une entrée dans le lexique de la langue.*

Le lemme présente toujours la flexion la plus neutre du paradigme. Pour les noms et les adjectifs, la flexion neutre combine dans la mesure du possible les valeurs « masculin » et « singulier », tandis que la flexion verbale neutre est l'infinitif.

Définition 13.1.12 (Affixe de dérivation)**.** *Un affixe de dérivation est un affixe qui permet de créer un nouveau lemme à partir d'un radical donné. Il peut s'agir d'un préfixe ou d'un suffixe.*

Définition 13.1.13 (Dérivation)**.** *La dérivation est le procédé linguistique de création de nouveaux lemmes à partir de radicaux auxquels sont ajoutés des affixes de dérivation.*

La dérivation peut être obtenue par préfixation (*fait* → *méfait*) ou par suffixation (*calme* → *calmement*). Les deux procédés peuvent être combinés (*gel* → *congélation*).

Lorsque plusieurs affixes sont simultanément nécessaires pour passer du radical au lemme, on parle de *parasynthèse*. C'est le cas, par exemple, pour *embrigader*, formé directement à partir de *brigade*, alors que *brigader** et (une) *embrigade** n'existent pas.

Deux types d'affixes existent. Les *affixes sémantiques*, qui permettent de créer des dérivés de sens différent mais de même nature (*fait* → *méfait*), et les *affixes lexicaux*, qui permettent de créer des dérivés de nature différente (*président* → *présidentiable*) ou de genre différent (*bon* → *bonne*).

Les affixes de dérivation et les flexions sont *a priori* différents. Il faut cependant considérer que certains suffixes sont à la fois des flexions et des affixes de dérivation. C'est le cas, par exemple, de l'infinitif (*don* → *donner*).

Remarque. La Définition 13.1.2 mentionne que le morphème est une unité abstraite. Qu'il s'agisse d'un radical ou d'un affixe, le morphème, étant constitué de phonèmes, est donc susceptible de se réaliser différemment selon la forme lexicale à laquelle il participe. Les différentes réalisations d'un morphème sont les *allomorphes*.

On comparera par exemple *déplacer* et *désorganiser*, *tenons* et *tiennent*, *irons* et *allons*, *chantent* et *vont*. Dans certains cas, il arrive même que les morphèmes fusionnent (dans *soit*, *soi* correspond à « être+conditionnel+présent »), ou disparaissent (le nombre n'apparaît pas dans *poids*, qui peut pourtant être singulier ou pluriel).

13.2 Choix d'implémentation d'eLite

Nous ne détaillons, dans cette section, que les éléments importants qui concernent la structure du système.

13.2.1 Structure de données et unité linguistique

La structure de données au travers de laquelle les modules d'eLite communiquent s'appelle la DLS, pour *Data Layer Structure*. Notre DLS (cf. Figure 13.1) est inspirée de la MLDS [3] proposée dans le cadre du projet Festival (Black *et al.* 1997).

Les trois premiers niveaux de la DLS sont créés ou modifiés par les modules d'analyse linguistique. Il s'agit des tokens, des words et des unités linguistiques. Les tokens et les words sont construits par le pré-processeur, tandis que les unités linguistiques sont construites par l'analyseur morphologique.

Note 13.2.1. Nous employons des termes anglais pour les deux premiers niveaux de la DLS afin d'éviter toute confusion avec les concepts utilisés en morphologie.

Définition 13.2.1 (Token)**.** *Le* token *est le premier niveau de la DLS, constitué soit d'une suite de caractères délimitée par des espaces ou des signes de ponctuation, soit d'un signe de ponctuation seul.*

[3] MLDS : *Multi Layers Data Structure*.

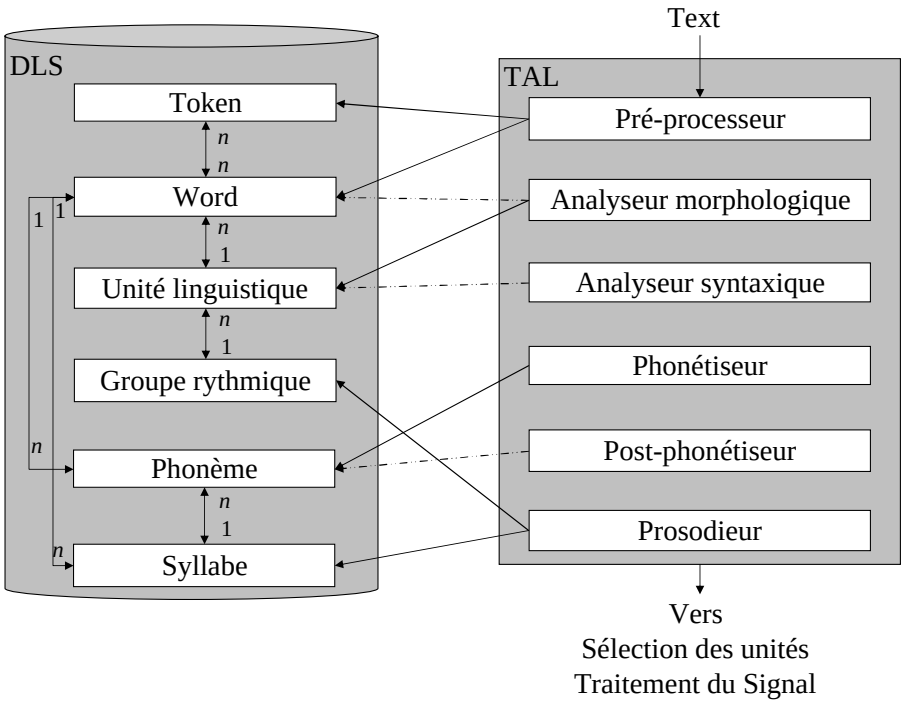


FIG. 13.1: Les modules du TAL et la structure de la DLS. Un trait plein entre un module et un niveau indique que le module crée le niveau, un trait en pointillés, que le module modifie le niveau

Définition 13.2.2 (Word)**.** *Le* word *est le second niveau de la DLS, constitué soit d'une suite de caractères alphabétiques, soit d'un symbole ou d'un signe de ponctuation seul. Un token peut donc recouvrir plusieurs words.*

Définition 13.2.3 (Unité linguistique)**.** *L'*unité linguistique *est le troisième niveau de la DLS. Linguistiquement, elle forme un tout, parce qu'elle a une unité de sens. L'unité linguistique est constituée d'un ou de plusieurs words.*

La totalité de la structure d'eLite repose sur la notion fondamentale d'*unité linguistique*. Telle que définie en 13.2.3, l'unité linguistique de base est évidemment le mot, au sens de la Définition 13.1.6. Il peut également s'agir d'un mot composé dont les constituants sont séparés par des espaces, des apostrophes ou des tirets, ou d'un acronyme dont les lettres sont séparées par des points. Enfin, ce peut être une unité complexe formée de caractères alphabétiques et de symboles, comme une URL, un numéro de téléphone, un nombre ou une unité de mesure. La Table 13.1 présente l'ensemble des unités linguistiques reconnues par eLite.

L'objectif de l'unité linguistique est de simplifier la tâche de l'analyseur syntaxique, en lui proposant d'analyser des catégories plus discriminantes, parce que de plus haut niveau.

En présence d'une URL par exemple, l'analyse syntaxique ne voit que la catégorie URI [4] correspondant à l'unité linguistique, au lieu d'une longue suite de catégories correspondant aux mots ou au symboles qui forment l'URL. L'unité linguistique, dans la structure de données d'eLite, est donc simplement une catégorie syntaxique.

Bien que l'unité linguistique soit une catégorie syntaxique et qu'elle regroupe parfois plusieurs words, chaque word conserve sa propre catégorie syntaxique. eLite est donc un système de synthèse pourvu de deux niveaux de catégories syntaxiques. Pour les distinguer, la catégorie syntaxique du word est qualifiée de *nature*. La Table 13.2 en donne un exemple.

Note 13.2.2. Les natures et catégories qui illustrent les différents exemples que nous donnons dans ce chapitre sont celles que nous employons dans notre analyseur. L'Annexe 1 en donne une description détaillée.

Ce double niveau permet à chaque module de recourir à l'information qui lui convient le mieux. Par exemple, l'analyseur syntaxique travaille au niveau des unités linguistiques afin d'avoir une vue d'ensemble de la phrase, tandis que le phonétiseur reste au niveau des natures, puisque la phonétisation d'un mot est initialement produite hors contexte.

Les unités linguistiques présentent un autre avantage. L'introduction de ce concept dans le système a été l'occasion de spécialiser les traitements réalisés par les différents modules *en fonction de* l'unité linguistique à traiter.

13.2.2 Données et langues

L'ensemble des données utilisées par eLite sont externalisées, de manière à rendre le système indépendant de la langue.

Les algorithmes des différents modules ne permettent cependant pas de traiter n'importe quelle langue, étant donné qu'eLite est principalement prévu pour gérer les caractéristiques linguistiques des langues flexionnelles [5], comme les langues romanes et l'anglais. Les langues actuellement traitées par eLite sont le français et l'anglais.

13.2.3 Rôle dans le développement d'eLite

Au niveau des réflexions préliminaires que nous venons de détailler, nous avons proposé le concept d'unité linguistique et l'idée de spécialiser le processus en fonction du type de l'unité.

En ce qui concerne les modules d'eLite, nous avons complètement développé l'analyseur syntaxique et le post-phonétiseur. Nous avons en outre proposé l'algorithme de l'analyseur morphologique, mais sans l'implémenter.

[4] URI : *Uniform Resource Identifier*. Il s'agit d'une chaîne de caractères identifiant une ressource Web (physique ou abstraite) dont la syntaxe respecte une norme W3C. Voir `www.w3.org/Addressing/`.

[5] La notion de *langue flexionnelle* est détaillée en Section 13.4 et repose sur la Définition 13.1.9.

Type	Exemples
Mot	mangera, ils, pré-maman
Mot composé	pomme de terre, c'est-à-dire, aujourd'hui
Acronyme	A.S.B.L., O.T.A.N
Ponctuation	. ! ? ; : - . . .
Date	01/02/2002, 30/06/75
Heure	8h 10min 30s, 10 :45
Téléphone	010/24.38.97, +33 3 27 33 34 54
Nombre	10.000.000, -10.045,43e-43
Montant	$40000, -10.000 EUR, +43,433.76 USD
Mesure	25 km/h, 10.343,45 N/m2
URI	www.cuisiner.fr/index.php ?x=10 10.108.55.9 john.smith@foo.co.uk

TAB. 13.1: Unités linguistiques détectées

Token	Word	(Nature)	Unité linguistique
C'est	c'	PRONDEM	PRONDEM
	est	PRED	PRED
la	la	DET	DET
pomme	pomme	NOUN	NOUN
de	de	PREP	
terre	terre	NOUN	
vendue	vendue	PARTPASSE	PARTPASSE
sur	sur	PREP	PREP
http ://www.vente.be	http	ACRONYM	URI
	:	SYMBOL	
	//	SYMBOL	
	www	ACRONYM	
	.	SYMBOL	
	vente	NOUN	
	.	SYMBOL	
	be	ACRONYM	
.	.	ENDPUNCT	ENDPUNCT

TAB. 13.2: double niveau d'analyse. Au sein d'une unité linguistique, chaque word conserve sa nature

13.3 Pré-processeur

L'ensemble des traitements réalisés par le pré-processeur sont représentés sous la forme d'expressions régulières.

Note 13.3.1. Il faut remarquer que le pré-processeur a été réalisé avant que notre bibliothèque et notre compilateur ne soient développés. Les expressions régulières du pré-processeur ont de ce fait été écrites selon le formalisme accepté par LEX, qui est un générateur d'analyseurs lexicaux : LEX prend en entrée des tables d'expressions régulières et des morceaux de codes associés, à partir desquels il génère un code C ou C++ à compiler. Pour de plus amples informations concernant ce logiciel, nous renvoyons le lecteur au site suivant : dinosaur.compilertools.net/.

Il est cependant important de signaler que le chercheur qui a travaillé au développement du pré-processeur a trouvé le moyen d'extraire, du code généré, les données propres à la langue. Or, le code généré par LEX ne diffère d'une langue à l'autre que par les données qu'il contient. Cette extraction des données du code permet donc au code du pré-processeur d'être indépendant de la langue. Comme les autres modules, le pré-processeur charge ses données au moment de l'initialisation.

Le rôle du pré-processeur est de diviser le texte en tokens. De la liste des unités linguistiques présentée en Table 13.1, toutes correspondent à un token reconnu par le pré-processeur, sauf les mots composés. Ceux-ci sont assimilés aux mots et seront détectés par la suite par l'analyseur morphologique. La liste des tokens est donc la suivante :

- Lexème
- Acronyme
- Ponctuation
- Date
- Heure
- Téléphone
- Nombre
- Montant
- Mesure
- URI

Note 13.3.2 (Lexème)**.** Le token « Lexème » est désigné de la sorte parce que les formes qu'il recouvre, constituées exclusivement de caractères alphabétiques, de l'apostrophe ou du tiret, correspondent toujours à une ou plusieurs formes lexicales telles que définies en 13.1.5. Ce token est donc toujours au moins constitué d'un lexème au sens défini en 13.1.3. Il arrive

donc qu'un token lexical recouvre plus d'une forme lexicale. Par exemple, *c'est* et *dit-on* sont des tokens de type lexical, constitués chacun de deux mots.

Dans la suite de ce document, nous employons de manière indifférenciée *lexème*, *token lexical* et *token de type lexical*.

Un token détecté est automatiquement segmenté en words. La segmentation réalisée respecte strictement la Définition 13.2.2. S'il y a eu « abus » de la segmentation, le problème sera résolu par l'analyseur morphologique.

Il faut ajouter que le pré-processeur détecte également les abréviations (*etc.*, *prof.*, *Mme*). Cependant, le token est étiqueté « lexical », parce que le word correspondant est créé avec la forme réécrite de l'abréviation (*prof.* → *professeur*).

Le pré-processeur profite de la segmentation qu'il réalise pour supprimer les caractères parasites, comme les espaces ou les caractères vides de sens. La DLS ne conserve donc pas la présentation originale du texte analysé.

Il effectue également une détection sommaire de la mise en page du document. Celle-ci se borne pour l'instant à repérer les titres (chiffres numériques ou romains suivis d'un point en début de ligne), les énumérations (points, astérisques ou tirets en début de ligne) et les paragraphes (multiples sauts de lignes), ce qui permet d'identifier des fins de phrase non marquées par un symbole de ponctuation et d'insérer des pauses de longueur adaptée dans la prononciation du texte.

13.4 Analyseur morphologique

L'objectif d'un analyseur morphologique est initialement de produire une analyse... morphologique. Cependant, de l'ensemble des tokens détectés par le pré-processeur, seuls le lexème et certaines parties du l'URI demandent un traitement de ce type. Les autres tokens se contentent d'un accès à un dictionnaire dédié.

L'analyseur morphologique tient compte de ces différences au niveau de son algorithme général : il itère sur les tokens, et pour chaque token, en teste le type et lance le traitement approprié.

Cette section commence par décrire le cœur de l'analyseur : l'analyse flexionnelle des formes lexicales. Elle présente ensuite les trois sous-modules de l'analyseur : le traitement des lexèmes, des URIs et des autres tokens.

13.4.1 Description de l'analyse flexionnelle

eLite étant prévu pour l'analyse de langues flexionnelles, l'analyse morphologique peut se limiter aux traitements nécessaires à ces langues. L'objectif en synthèse de la parole n'est certainement pas de décomposer les mots d'un texte en morphèmes. Ce traitement

impliquerait de détecter les radicaux, les flexions et les affixes de dérivation, ainsi que d'identifier les unités abstraites derrière les allomorphes. Or, une telle décomposition apporterait une somme d'informations linguistiques inutiles au processus de synthèse dans l'état de l'art actuel.

Les seules informations qui intéressent *a priori* le processus de synthèse sont la nature et les traits grammaticaux des formes lexicales. Au niveau morphologique, la détection de ces informations demande donc tout au plus une analyse flexionnelle. L'analyse dérivationnelle peut donc être mise de côté.

13.4.1.1 Etat de l'art en analyse flexionnelle

Approche algorithmique. Pour une forme donnée, une analyse flexionnelle simple consiste à supprimer progressivement des terminaisons de plus en plus longues, à ajouter à chaque suppression des flexions neutres, susceptibles de construire un lemme, et à tester l'existence de la forme reconstruite dans un lexique de lemmes. Lorsqu'un lemme a été trouvé, on vérifie que sa nature autorise les transformations qui ont été appliquées.

Cette approche a été proposée par Winograd (1972). Elle utilise peu d'informations morphologiques : un lexique de lemmes et un lexique de flexions neutres associées à leurs natures. Elle est cependant surtout applicable aux langues à flexion pauvre, comme l'anglais [6], parce qu'elle implique de nombreux tests inutiles qui allongeraient exagérément le temps de traitement dans le cas d'une langue à flexion riche, comme le français.

Approche déclarative. Une autre approche (Pitrat 1983, Sabah 1988) est, elle, mieux adaptée au français. Elle utilise 3 lexiques :

1. Les lemmes.
2. Les classes flexionnelles.
3. Les groupes de flexions.

Le lexique des lemmes renseigne, pour chaque lemme, sa classe flexionnelle et ses radicaux. La classe flexionnelle est un lemme :

$$\texttt{lemme} \quad \texttt{classe} \quad \texttt{radical}_1 \quad \texttt{radical}_2 \quad \ldots \quad \texttt{radical}_n$$

Dans le lexique des classes flexionnelles, une classe contient un ou plusieurs modèles de flexion. Une classe qui s'applique à un nom ou un adjectif compte un ou deux modèles selon les genres et nombres applicables, tandis qu'une classe qui s'applique à un verbe contient autant de modèles qu'il y a de combinaisons {mode, temps}. Un modèle est décrit sur une ligne et présente le groupe de flexions et les radicaux à utiliser :

[6] Un verbe anglais compte au plus 8 formes, et très souvent seulement 3. A titre de comparaison, la majorité des verbes français comptent 37 formes, certains allant jusqu'à 40.

```
classe    modele_1  groupe_1  indice_radical_1  ...  indice_radical_n
          modele_2  groupe_2  indice_radical_o  ...  indice_radical_p
          ...
          modele_k  groupe_k  indice_radical_q  ...  indice_radical_r
```

Le lexique des groupes de flexions liste, pour chaque groupe, l'ensemble de ses flexions

```
groupe_1    flexion_1  ...  flexion_n
groupe_2    flexion_o  ...  flexion_p
```

La Table 13.3, extraite de (Boîte *et al.* 2000), donne un exemple d'entrées dans les trois lexiques pour les lemmes *tenir* et *venir*. Ces deux verbes appartiennent à la même classe flexionnelle. L'un des deux, *venir*, est posé comme la classe. L'autre, *tenir*, est dès lors introduit comme un membre de cette classe.

Lexique de lemmes

```
tenir      venir  tien  ten  tienn  tin  tîn
venir      venir  vien  ven  vienn  vin  vîn
...
```

Lexique des classes flexionnelles

```
venir      indicatif_present  ind_pre  1  1  1  2  2  3
           indicatif_imparfait  ind_imp  2  2  2  2  2  2
...
```

Lexique des groupes de flexions

```
ind_pre    s  s  t  ons  ez  ent
ind_pre    ais  ais  ait  ions  iez  aient
...
```

TAB. 13.3: Lexiques de l'analyse morphologique déclarative

Au contraire de l'approche précédente, celle-ci ne cherche dans une forme que les flexions connues de ses lexiques. Lorsqu'une flexion a été trouvée, le système vérifie que la racine existe. Dans l'affirmative, il recherche dans la classe toutes les instances de la flexion qui s'appliquent sur la même racine afin de produire toutes les désinences correspondantes.

Cette approche a cependant trois inconvénients :

1. La désinence n'est indiquée dans aucun lexique. C'est donc le système qui la produit en fonction de la position de la flexion dans le lexique. Le système n'externalise donc

pas complètement ses données. Par exemple, la désinence « indicatif, présent, 1, sg » pour *tiens* est déduite par le système de la position de la flexion dans le lexique.

2. L'approche nécessite la description de l'ensemble des allomorphes radicaux.
3. Par rapport au reste du traitement réalisé, la recherche des instances d'une flexion dans l'ensemble de la classe est une étape relativement lourde.

13.4.1.2 Analyse flexionnelle dans eLite

L'analyse flexionnelle dans eLite s'inspire de l'approche déclarative, mais tâche d'en éviter les inconvénients.

Les lexiques. Nous désirions que les lexiques contiennent directement les désinences, afin de libérer au maximum le système de ses données. L'objectif était que le système puisse proposer une désinence sans devoir tenir compte d'une quelconque information positionnelle.

Au lancement d'eLite, nous avons eu la possibilité d'obtenir les lexiques utilisés dans le cadre du synthétiseur Euler (Dutoit *et al.* 1998) et initialement développés dans le cadre du projet Morlex [7]. Deux lexiques contiennent l'ensemble des informations morphologiques :

1. Le lexique de lemmes (cf. Table 13.4). Chaque ligne du lexique de lemmes contient un lemme, sa nature et sa classe flexionnelle.
2. Le lexique des classes flexionnelles (cf. Table 13.5). Pour une classe flexionnelle donnée, le fichier contient autant de lignes que la classe contient de désinences. Le principe de ce lexique est de décrire la *terminaison à remplacer* pour obtenir un lemme à partir d'une forme fléchie : sur une ligne, le fichier contient la classe flexionnelle, la terminaison à supprimer, la terminaison remplaçante et la désinence.

   Dans son ensemble, cette approche est plus pragmatique : elle ignore les frontières morphémiques. La terminaison à remplacer peut donc être :

   - Egale à la flexion. C'est le cas des verbes de la conjugaison en « er » (cf. Table 13.5, classe flexionnelle numéro 6).
   - Supérieure à la flexion, comme pour le verbe *avoir* (cf. Table 13.5, classe flexionnelle numéro 1), dont l'irrégularité complique les traitements purement morphologiques.
   - Nulle, s'il n'y a pas de différence entre le lemme et la flexion (cf. le symbole $\emptyset$ dans la Table 13.5).

Note 13.4.1. Dans la suite de ce document, nous continuons néanmoins à parler de *flexion* pour plus de clarté.

[7] bach.arts.kuleuven.be/pmertens/morlex/.

beau	ADJECTIF	mf,sp,-x+,-elle+eau
bête	ADJECTIF	mf,sp,-s+,-+
...		
abandonnataire	NOUN	mf,sp,-s+,-+
abaque	NOUN	m,sp,-s+,-+
...		
abaisser	VERB	6
abattre	VERB	55
abcéder	VERB	10a

TAB. 13.4: eLite : lexique de lemmes

1	a	avoir	3,s,IdPr
1	eu	avoir	m,s,PtPa
...			
6	a	er	3,s,IdPa
6	ai	er	1,s,IdPa
...			
m,sp,-+,-+	∅	∅	m,s
m,sp,-+,-+	∅	∅	m,p
...			
m,sp,-s+,-+	s	∅	m,p
m,sp,-s+,-+	∅	∅	m,s
...			
mf,sp,-+,-esse+ès	∅	∅	m,p
mf,sp,-+,-esse+ès	esse	ès	f,s
mf,sp,-+,-esse+ès	esses	ès	f,p
mf,sp,-+,-esse+ès	∅	∅	m,s

TAB. 13.5: eLite : lexique des classes flexionnelles. Le symbole ∅ signifie que la terminaison est nulle

L'algorithme de l'analyse flexionnelle est présenté en Pseudocode 31. Cet algorithme nécessite une structure de données qui permette de détecter, en une seule étape, l'ensemble des flexions qui terminent une forme et l'ensemble des analyses qui y correspondent. La structure la plus appropriée est incontestablement le *trie*.

La structure de données. L'origine du terme *trie* est le terme anglais *retrieval* [8] qui, dans le domaine de l'informatique, prend le sens d'« extraction ». Le trie est un arbre.

Un arbre est un graphe orienté acyclique (cf. Figure 13.2, a). Chaque nœud de l'arbre peut posséder des transitions sortantes vers des nœuds que l'on appelle ses fils, et possède au plus une transition entrante, provenant d'un nœud que l'on appelle son père. Le nœud racine est le seul à ne pas posséder de transition entrante. Les nœuds terminaux n'ont quant à eux pas de transition sortante. Tout parcours de l'arbre commence au nœud racine et ne donne un résultat que s'il s'achève à un nœud terminal.

La fonction du trie est d'encoder un tableau associatif de paires {clef,valeur} (de la Briandais 1959, Fredkin 1960). Les clefs encodées par le trie sont classiquement des strings, qui étiquettent des séquences de transitions depuis le nœud racine à raison d'un caractère par transition. Le nœud atteint par le dernier caractère de la clef est terminal et contient la valeur associée à la clef. A partir du nœud racine, le parcours d'une string appartenant à l'arbre ne permet d'atteindre qu'un seul nœud. Contrairement à un arbre classique, les nœuds terminaux du trie peuvent de ce fait présenter des transitions sortantes, puisqu'une clef peut être le préfixe d'une clef plus longue (cf. Figure 13.2, b). Cette particularité du trie en fait la seule structure de données capable de détecter, en un seul parcours, l'ensemble des clefs qui préfixent une string donnée, et qui retourne l'ensemble des valeurs associées à ces préfixes. Le trie est de ce fait qualifié d'*arbre préfixe*.

Cette structure convient parfaitement pour encoder l'ensemble des flexions. Cependant, étant donné que le trie encode des préfixes,

- Les flexions doivent être inversées avant d'y être encodées.
- Toute forme lexicale à parcourir dans le trie doit être inversée avant le parcours.

De la sorte, on parcourt effectivement dans le trie l'ensemble des préfixes d'une string. Dans eLite, la valeur qui est mémorisée dans un nœud du trie de flexions pointe vers une paire qui associe :

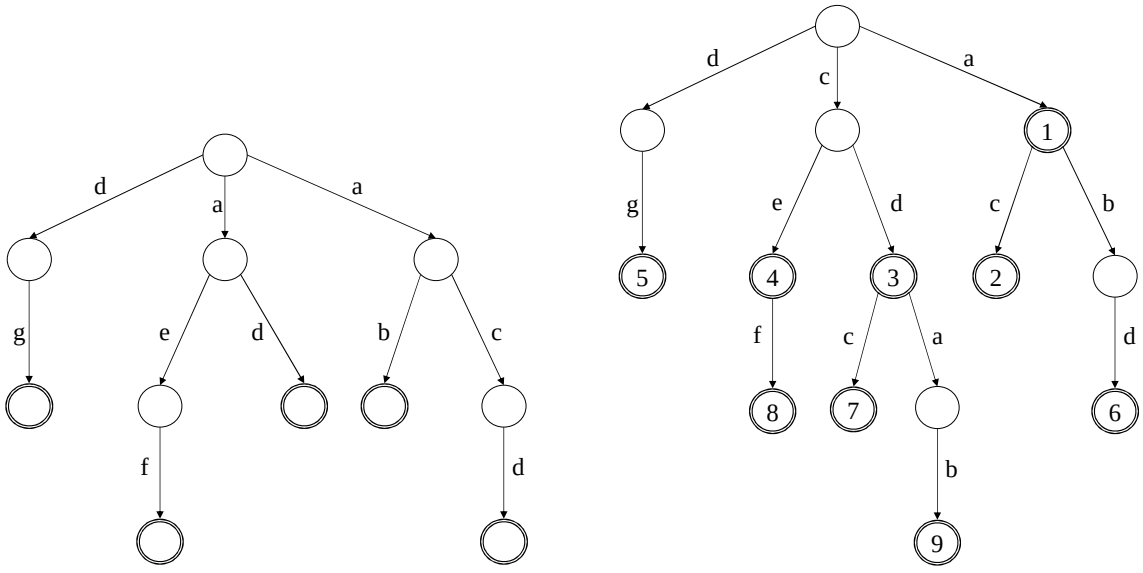
- La flexion elle-même. Ceci est nécessaire puisqu'elle aura été parcourue dans le trie sans être mémorisée.
- L'ensemble des triplets (classe flexionnelle, flexion neutre, désinence) qui correspondent à la flexion.

La Figure 13.3 en donne un exemple.

[8] Malgré cette origine, la tendance veut cependant que *trie* se prononce [trai], probablement pour garder la distinction avec le terme anglais *tree*.

Require : Une forme lexicale
Ensure : L'analyse flexionnelle de cette forme lexicale
1 : **Pour** une forme lexicale donnée,
2 : Trouver l'ensemble des flexions appartenant au lexique des classes flexionnelles.
3 : **Pour chaque** flexion trouvée,
3 : **Pour chaque** classe flexionnelle associée,
4 : Construire le lemme en remplaçant la flexion par la flexion neutre de la classe flexionnelle.
5 : **Si** la paire {lemme, classe flexionnelle} existe dans le lexique de lemmes,
6 : Ajouter la désinence aux analyses valides.

Pseudocode 31: Analyse morphologique des lexèmes



a. L'arbre classique *b. Le trie*

FIG. 13.2: Comparaison d'un arbre classique et d'un trie. Les nœuds terminaux sont représentés par des doubles cercles

Au moment du chargement du lexique de classes flexionnelles, deux structures sont donc initialisées : un vecteur de paires {flexions, ensemble de triplets} et le trie de flexions inversées. De la sorte, aucune recherche supplémentaire n'est nécessaire en cours de traitement. L'algorithme décrit précédemment est donc respecté.

Le parcours du trie de la Figure 13.3 avec la forme lexicale « chantent » inversée en « tnetnahc » retournera les paires pointées par les valeurs 1 ($\emptyset$), 6 (*-t*), 7 (*-nt*) et 8 (*-ent*).

Gestion de la casse et des formes désaccentuées. Les « fautes d'orthographe » les plus communes sont certainement la différence de casse et l'absence d'accentuation. Il est très fréquent, par exemple, de rencontrer des phrases comme *LES ELEVES SONT BIEN ELEVES*.

En l'absence d'un véritable système de correction, le système autorise une recherche « relâchée » : lorsque ce mode de recherche est activé, le parcours dans le trie et la recherche du lemme dans le lexique acceptent les différences d'accentuation et les différences de casse. La recherche de *ELEVES*, par exemple, détectera les formes accentuées {*élevés*,`PARTPASSE`} et {*élèves*,`NOUN|VERB`}.

La recherche relâchée n'inclut cependant pas de mécanisme de pondération des formes trouvées : la structure d'eLite ne contient aucun mécanisme de mémorisation des poids qui seraient attribués par l'analyse. Pour cette raison, ce mode n'est activé que si la recherche classique n'a donné aucun résultat, et les candidats trouvés sont équiprobables.

13.4.2 Traitement des tokens lexicaux

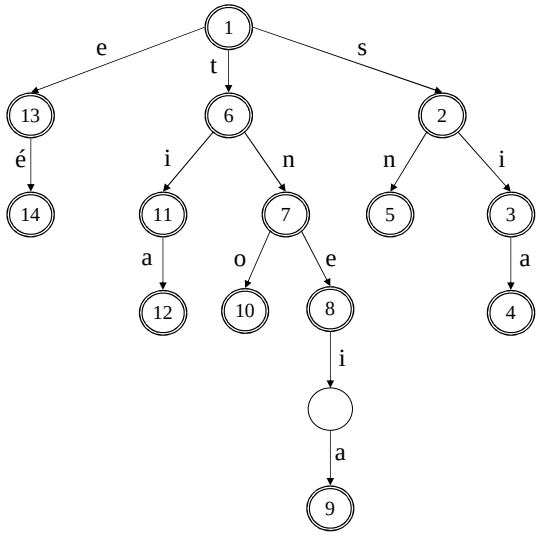
13.4.2.1 Etat des lieux

Au sortir du pré-processeur, certains tokens lexicaux ne correspondent pas à des unités linguistiques. Il s'agit des mots composés et des séquences de lettres contenant des apostrophes et des tirets. La Table 13.6 illustre ce fait.

Cette limite est cependant justifiée, parce que le pré-processeur, sans traitements supplémentaires, ne peut prendre de décision *a priori*. Comme le montre la Table 13.6, il arrive que plusieurs tokens correspondent à une seule unité (*pomme de terre*). L'apostrophe indique parfois un mot composé (*aujourd'hui*), mais peut également noter l'élision du terme de gauche (*c'est* pour *ce est*) ou de droite (en anglais, *it's* pour *it is*). Le trait d'union indique un mot composé (*après-midi*) ou l'enclise du pronom personnel (*dit-on*).

En outre, il est évident que de nombreux mots composés se créent tous les jours et n'appartiennent pas au lexique. Afin de faciliter la tâche de l'analyseur syntaxique en limitant le nombre des unités linguistiques, il peut être intéressant de les recomposer également. Il existe 4 types de mots composés :

1. Les composés unifiés, qui se sont soudés dans l'écriture. Si la plupart des composés unifiés appartiennent de longue date à la langue (*portefeuille*, *gendarme*, *pourcent*), il



1 : "", { ("m,s,-+,-+", "", "m,s"), ("mf,p,-+,-+", "", "m,p"), ("mf,p,-+,-+", "", "f,p") }
2 : "s", { ("25", "tir", "1,s,IdPr"), ("f,sp,-s+,-+","", "f,p") }
…
14: "ée", { ("6", "er", "f,s,PtPa"), ("13", "éer", "1,s,IdPr"), ("65", "aître", "f,s,PtPa") }

FIG. 13.3: Un trie de flexions. Les flexions sont inversées dans le trie. Les valeurs pointent vers des paires {flexions, ensemble de triplets}

Token	**Word**
pomme	pomme
de	de
terre	terre
aujourd'hui	aujourd
	'
	hui
c'est	c
	'
	est
après-midi	après
	-
	midi
dit-on	dit
	-
	on

TAB. 13.6: Résultat du traitement du pré-processeur sur différents tokens lexicaux

arrive néanmoins que de nouveaux composés s'unifient. Certains écrivent par exemple *préprocesseur*...

2. Les composés détachés, comme *pomme de terre*.
3. Les composés à apostrophe, comme *aujourd'hui* ou *entr'aide*. Le procédé marque l'élision d'un schwa.
4. Les composés à traits d'union, comme *après-midi*.

Les outils dont nous disposions dans cette version de l'analyseur ne nous permettaient pas de gérer aisément les composés unifiés, au vu des traitements lourds que l'unification aurait imposés à l'analyseur. Les mots composés détachés demandent quant à eux de l'information d'ordre sémantique, ce qui dépasse largement l'objectif de l'analyse dans le cadre de la synthèse. Un parcours de corpus de textes nous a en outre fait constaté que les composés à apostrophes ne semblent pas fréquents, au contraire des composés à traits d'union, pour lesquels nous avons rencontré de nombreux cas inconnus de nos lexiques, qu'il s'agisse de néologismes ou non : *pré-processeur*, *sur-couver*, *essai-erreur*, etc.

Au lieu de compléter nos lexiques de mots composés à traits d'union rencontrés dans quelques corpus inévitablement incomplets, nous avons préféré intégrer dans l'analyseur une procédure de détection de ces mots composés. L'idée est simple. Nos lexiques contiennent de nombreux exemples valides de composés à traits d'union. Nous les avons donc analysés, de manière à proposer une liste de suite de natures, séparées par des traits d'union, susceptibles de constituer un mot composé. La liste des suites de natures, séparées par des traits d'union, dont nous acceptons la recomposition en une seule unité linguistique, est présentée en Annexe 2.

13.4.2.2 Algorithme général

Cet état des lieux met en évidence la nécessité des traitements suivants :

1. Réaliser une analyse flexionnelle des formes lexicales telle que décrite en Section 13.4.1.
2. Gérer les formes pourvues d'apostrophes ou de traits d'union. Soit les formes sont toutes à réunir en une seule (*aujourd'hui*, *après-midi*), soit une partie des formes sont à réunir (*c'* et *est*, *it* et *'s*), soit toutes les formes restent séparées (*dit*, -, *on*). Ceci demande des tests de recomposition sur les formes, chaque test incluant une analyse flexionnelle permettant de valider ou d'invalider la forme recomposée. Les formes pourvues de traits d'union nécessitent en outre un test de recomposition sur la base des natures.
3. Recomposer les mots composés détachés, qui ont été séparés entre plusieurs tokens du fait de la présence de l'espace entre les composants. Cette étape permet de construire une seule unité linguistique à partir de plusieurs tokens, mais l'analyse flexionnelle de chaque forme reste nécessaire. Cette recomposition ne doit donc être réalisée qu'*après* l'analyse flexionnelle des composants.

Require : on part d'un token lexical
Ensure : à partir de ce token, tous les tokens lexicaux contigus sont analysés et au besoin recomposés

```
1 :  Tant que le token courant est de type lexical,
2 :    Un compteur de traits d'union est mis à 0.
3 :    Pour chaque word du token courant,
4 :      Si le word courant est un trait d'union,
5 :        Incrémenter le compteur de traits d'union.
6 :      Sinon Si le word courant n'est pas une apostrophe,
7 :        Si le word courant est précédé ou suivi d'une apostrophe ou d'un trait d'union,
8 :          Tant qu'une recomposition n'a pas été testée et qu'il n'y a pas de résultat,
9 :            Tester la validité d'une nouvelle recomposition.
10 :         Si une recomposition est validée,
11 :           Remplacer le word courant par la recomposition valide.
12 :           Supprimer du token courant les words devenus inutiles.
13 :           Si le déclencheur de la recomposition est un trait d'union, décrémenter le
               compteur de traits d'union.
14 :       Si le word courant n'est pas encore analysé, exécuter l'analyse flexionnelle.
15 :       Si le word courant n'est pas encore analysé, déclencher la procédure dédiée
           aux mots inconnus.
16 :   Si le compteur de traits d'union est supérieur à 0, rechercher les composés sur
       la base des natures.
17 :   Passer au token suivant.
18 : Dans cette suite de tokens, rechercher les composés détachés.
19 : Dans cette suite de tokens, supprimer les words dont la nature fait partie de
     la Stop List.
```

Pseudocode 32: Analyse morphologique du token lexical

L'algorithme se déduit des traitements à réaliser et est présenté en Pseudocode 32. Cet algorithme demande quelques commentaires :

Compteur. L'algorithme inclut un compteur de traits d'union (ligne 2), qui permet de détecter la présence, dans un token, d'au moins un trait d'union (ligne 5). Si le compteur est positif, la procédure de recomposition sur les natures est exécutée (ligne 16). Notons qu'en cours de traitement, le compteur peut être décrémenté, si un trait d'union a pu être rattaché aux words qui l'entourent par recomposition (ligne 13).

Recomposition lexicale. Les tests de recomposition lexicale (lignes 7–13) sont les suivants :

1. *Le trait d'union.* Son traitement est fort simple : le trait d'union *doit* unir les termes qui l'entourent. Il suffit donc de prendre la plus longue suite de words séparés par des traits d'union, de les concaténer, et de les analyser.
2. *L'apostrophe.* Son traitement est plus complexe. Si l'apostrophe suit le word courant, le système doit d'abord tester la recomposition incluant le deuxième word suivant (*aujourd'hui*). S'il échoue, le système teste la recomposition du word courant et de l'apostrophe (*c'*). Par contre, si l'apostrophe précède le word courant, le seul test de recomposition inclut l'apostrophe et le word courant (*'s*).

Mots inconnus. La procédure dédiée (ligne 15) attribue à un mot inconnu les 4 natures considérées comme ouvertes, dans le sens où elles acceptent des néologismes : `ADJ`, `ADV`, `NOUN`, `VERB`. Dans cette version du système, l'attribution se fait sans analyse flexionnelle. Ce choix a été guidé par l'hypothèse qu'il faut laisser à l'analyse syntaxique la possibilité de choisir parmi un large ensemble de possibilités, plutôt que de réduire ces possibilités du fait de quelques tests morphologiques, qui peuvent s'avérer inexacts dans le cas de termes étrangers.

Composés détachés. La recherche des composés détachés (ligne 18) utilise un lexique de recomposition, qui contient un composé par ligne. Les informations fournies sont les composants lexicaux, la catégorie syntaxique de l'unité et les natures des composants lexicaux. La Table 13.7 en donne quelques exemples.

Composants lexicaux	Catégorie syntaxique	Natures des composants
`assez peu de`	`DETIND`	`ADVDEG ADVDEG PREP`
`la plupart d' entre`	`DETIND`	`DET NOUN PREP PREP`
`chefs d' oeuvre`	`NOUN`	`NOUN PREP NOUN`
`à demi - mot`	`ADV`	`PREP ADJ TIRET NOUN`

TAB. 13.7: Lexique de recomposition

Le principe est de rechercher, dans la suite de tokens lexicaux, la plus longue suite de words constituant un mot composé. Pour ce faire, l'algorithme utilise une fenêtre initialisée à la taille du plus long composé du lexique, et réduit cette fenêtre jusqu'à ce qu'un résultat soit trouvé ou que la fenêtre soit de dimension 1. Actuellement, le plus long composé de notre lexique contient 7 composants : *au - fur et à mesure que.*

Stop list. La suppression des words dont la nature appartient à une *stop list* (ligne 19) a été ajoutée à l'algorithme afin de faire disparaître de la DLS tout word dont la catégorie n'intéresse pas le processus qui a fait appel à l'analyseur morphologique. La *stop list* est

contenue dans un lexique chargé par le système. Ceci intéressait tout particulièrement un module d'extraction d'information dans lequel notre analyseur a été intégré. Dans le cadre d'eLite, nous employons cette étape pour supprimer les traits d'union, qui n'ont plus d'intérêt pour la suite du processus.

13.4.3 Traitement des tokens URI

Le token URI est facilement détecté. La syntaxe d'une URI (URLs, mails, IPs [9]. numériques) peut être décrite à l'aide d'expressions régulières, pour autant qu'elle respecte le protocole w3c [10]. De manière fort générale, une URI se divise en parties délimitées par une liste fixe de symboles (:, `//`, `@`, `?`, `~`, etc.). Certaines parties de l'URI sont réservées à des mots clefs (`http`, `ftp`, `mailto`, `www`, `com`, `.org`, `be`, `fr`, etc.) listés par le protocole. Les autres parties sont libres, par contre, de présenter un vocabulaire plus que varié :

1. L'ensemble des formes lexicales rencontrées dans les lexiques peuvent y apparaître, mais désaccentuées. Dans une URI, les formes *élèves* et *élevés*, par exemple, apparaîtront sous la même forme *eleves*.
2. Une forme peut être constituée de plusieurs formes lexicales agglutinées : `http://www.commentcamarche.net`.
3. Les formes lexicales peuvent appartenir à n'importe quelle langue, dialecte ou idiome.
4. Les noms propres de personnes, de lieux ou d'entités sont évidemment fréquents, de même que les acronymes.
5. En pratique, n'importe quelle suite de caractères alphabétiques est acceptée, pour autant que la longueur totale de l'URI ne dépasse pas une certaine limite. Internet Explorer, par exemple, n'accepte pas plus de 2 083 caractères [11]...

Dans cette version de l'analyse morphologique, l'accent n'a pas été mis sur la gestion des formes agglutinées : des formes telles que `commentcamarche` sont simplement considérées comme hors-vocabulaire. Seules les formes lexicales simples sont gérées. L'analyse flexionnelle est donc réalisée comme nous l'avons décrite en Section 13.4.1.2, si ce n'est que le trie est toujours parcouru en mode « recherche relâchée », de manière à gérer l'absence constante d'accentuation. Ceci permet au moins de proposer des analyses pour des termes comme *eleves*.

13.4.4 Traitement des autres tokens

Les autres tokens ont été détectés par le pré-processeur, parce qu'ils présentent une syntaxe régulière, comme les URIs. Par contre, ils se construisent sur un lexique réduit. De

[9] Internet Protocol
[10] Description du protocole URI : www.w3.org/Addressing/.
[11] Source : support.microsoft.com/kb/208427.

ce fait, le rôle de l'analyseur morphologique se limite à attribuer à chaque partie du token sa nature, lue dans un lexique spécialisé, et à créer une unité linguistique dont la catégorie correspond à la valeur du token. La Table 13.8 donne un exemple du traitement réalisé sur un token de type « Mesure ». La table met en évidence le fait que le pré-processeur a réécrit les nombres en lettres. Cette réécriture est valable pour tous les types de tokens, autres que le lexème et l'URI.

Token	**Word**	**Nature attribuée**	**Unité linguistique créée**
38m2	trente	NUM	UNIT
	huit	NUM	
	mètre	NOUN	
	carré	NOUN	

TAB. 13.8: Gestion d'un token Unité

13.5 Analyseur syntaxique

13.5.1 Introduction

Comme illustré par la Table 13.9, au sortir de l'analyseur syntaxique, chaque word a été associé à une seule unité linguistique. Une unité linguistique peut par contre contenir plusieurs words. En terme d'analyse, le word peut proposer plusieurs natures et l'unité linguistique, plusieurs catégories syntaxiques.

Token	**Word**	**Nature attribuée**	**Unité linguistique créée**
c'est	c'	PRONDEM	PRONDEM
	est	NOUN|AUX|PRED	NOUN|AUX|PRED
les	les	PRONPERCD|DET	PRONPERCD|DET
poules	poules	NOUN	NOUN
du	du	DETPREP	DETPREP
couvent	couvent	NOUN|VERB	NOUN|VERB
qui	qui	PRONREL|PRONINT	PRONREL|PRONINT
sur-couvent	sur	PREF|PREP	NOUN|VERB
	couvent	NOUN|VERB	

TAB. 13.9: Entrée de l'analyseur syntaxique

Dans eLite, l'analyseur syntaxique a deux tâches. Premièrement, il doit déterminer la catégorie syntaxique à conserver pour chaque unité linguistique, à l'aide d'une analyse syn-

taxique. Deuxièmement, il doit choisir la nature de chaque word, en fonction de la catégorie retenue au niveau de l'unité linguistique. La Table 13.10 en donne une illustration.

Token	**Word**	**Nature attribuée**	**Unité linguistique créée**
c'est	c'	PRONDEM	PRONDEM
	est	PRED	PRED
les	les	DET	DET
poules	poules	NOUN	NOUN
du	du	DETPREP	DETPREP
couvent	couvent	NOUN	NOUN
qui	qui	PRONREL	PRONREL
sur-couvent	sur	PREF	VERB
	couvent	VERB	

TAB. 13.10: Objectif de l'analyseur syntaxique

Dans le cadre d'un système de synthèse, les traitements syntaxiques nécessaires sont donc fort limités. En outre, notre objectif en développant eLite était de le doter d'un analyseur :

1. Robuste : l'analyse *doit* donner un résultat, afin de rendre possible les traitements suivants. Le système ne peut laisser ambiguë la sortie de l'analyse morphologique, et ne peut choisir une analyse *par défaut* sous prétexte que la phrase a été impossible à analyser.
2. Multilingue : l'analyseur syntaxique doit être rapidement portable dans une autre langue.

Ci-dessous, nous commençons par dresser un bref état de l'art de l'analyse syntaxique, avant de présenter l'analyse syntaxique d'eLite.

13.5.2 Etat de l'art en analyse syntaxique

13.5.2.1 Les systèmes experts

Ce point n'a pas la prétention de rendre justice à la qualité des travaux linguistiques réalisés par les chercheurs du domaine. Il tient simplement à attirer l'attention du lecteur sur les caractéristiques majeures des approches linguistiques. Pour de plus amples informations, le lecteur intéressé se reportera utilement à la bibliographie non exhaustive suivante : (Harris 1962, Gross 1984, Roche 1993, Tapanainen & Jarvinen 1994, Ait-Mokhtar & Chanod 1997, Voutilainen 2001, Fairon *et al.* 2005).

Quelle que soit la méthode d'analyse syntaxique appliquée, l'objectif de l'analyse est de déterminer la *structure* syntaxique de la phrase analysée : de manière générale, des règles déterminent les regroupements lexicaux qui donnent des syntagmes valides, et les

regroupements de syntagmes qui permettent de remonter jusqu'à un axiome « S » qui représente la racine de toute phrase valide (cf. Figure 13.4). Pour améliorer le système, les chercheurs ont rapidement augmenté leurs analyses syntaxiques d'informations lexicales et sémantiques, entre autres au moyen de structures de traits qui décrivent le type de contexte syntaxique et/ou sémantique dans lequel un mot donné peut être utilisé.

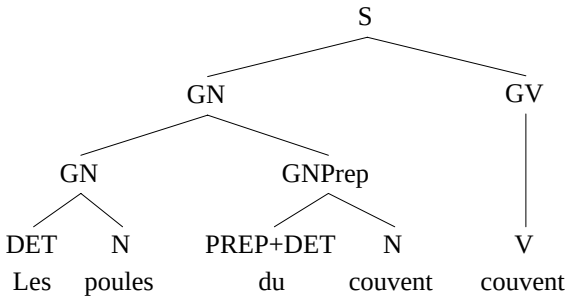


FIG. 13.4: Arbre d'analyse syntaxique

On peut immédiatement identifier les limites de cette approche :

1. Certaines phrases autorisent plusieurs analyses syntaxiques (cf. Figure 13.5). Choisir automatiquement la bonne analyse demande dès lors des connaissances au moins sémantiques, mais fréquemment pragmatiques. Or, l'acquisition et la gestion de connaissances pragmatiques est un défi qui n'a actuellement été qu'abordé par la recherche.

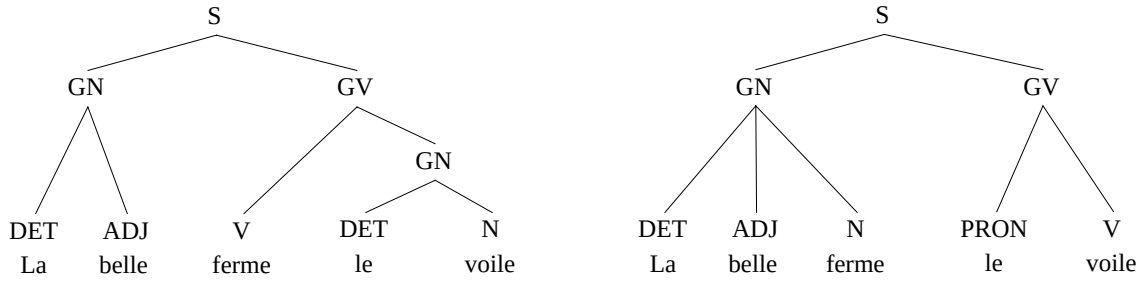


FIG. 13.5: Analyse syntaxique ambiguë

2. Certaines phrases ne permettent pas d'analyse arborescente du fait de la présence d'une rupture de construction : il s'agit d'une rupture dans la construction d'une « phrase canonique », constituée d'une interruption, puis d'une reprise adoptant une autre structure syntaxique [12]. Ce phénomène est assez fréquent dans la correspondance [13], dans

[12] valibel.fltr.ucl.ac.be/julibel.htm.
[13] Tout particulièrement le courrier électronique.

les ébauches ou lorsque le texte est une retranscription d'un conversation orale ou qu'il « mime » l'oralité :

« *Le nez de Cléopâtre, s'il eût été plus court, toute la face de la terre aurait changé.* » [14] (anacoluthe)

« *L'ai reconnue tout de suite, les yeux de son père.* » [15] (ellipse)

« *Je pense aux... que les hommes vont suivre.* » (interruption)

L'analyse syntaxique, même augmentée de structures de traits, va buter sur ce type de phrases.

En outre, c'est un travail long et délicat que de proposer un modèle syntaxique qui couvre la totalité de la norme d'une langue, et qui soit capable de gérer les phrases ambiguës et les ruptures de construction. Pour le français, un système linguistique commercial a été développé : il s'agit du logiciel Antidote, qui propose une correction *supervisée* [16]. Ce système est très performant, mais est le résultat de 10 années de développements réalisés par une équipe comptant de nombreux experts en linguistique française. Il est de ce fait difficile de concevoir rapidement un système multilingue qui propose une telle qualité.

13.5.2.2 Les modèles de langue

Etant donné une suite de m mots $W = (w_1, w_2, \ldots, w_m)$, un modèle de langue tâche de déterminer la meilleure suite de catégories syntaxiques $\hat{T} = (t_1, t_2, \ldots, t_m)$ parmi toutes les suites $\{T_1, T_2, \ldots, T_{l-1}, T_l\}$, où l correspond au nombre de combinaisons possibles des catégories attribuables aux mots w_1 à w_m :

$$\hat{T} = \arg\max_T P(T|W) \tag{13.5.2.1}$$

ce qui, par le théorème de Bayes (Boîte *et al.* 2000), se résout classiquement par :

$$\hat{T} = \arg\max_T \frac{P(W|T)P(T)}{P(W)} \tag{13.5.2.2}$$

Etant donné que W est constant quel que soit T, l'équation peut se réduire à :

$$\hat{T} = \arg\max_T P(W|T)P(T) \tag{13.5.2.3}$$

Note 13.5.1. Dans le modèle de langue tel qu'il se présente en Equation 13.5.2.3, nous posons que $P(W|T)$ est le *modèle lexical* et que $P(T)$ est le *modèle syntaxique*.

Les deux modèles s'estiment classiquement à l'aide d'un corpus de phrases où chaque mot est associé à sa catégorie syntaxique. Le modèle est donc facilement portable,

[14] Blaise Pascal, *Pensées*. Ce recueil est l'ébauche d'une œuvre qui n'a jamais vu le jour.

[15] James Joyce, *Ulysse*.

[16] Antidote : `www.druide.com/a_description.html`. Ce logiciel s'intègre dans les traitements de texte du commerce.

puisqu'il est assez aisé de trouver un corpus étiqueté pour de nombreuses langues. Ce modèle est en outre robuste, étant donné qu'il produit systématiquement une analyse, quelle que soit la phrase.

Quel que soit le système, le modèle de langue présenté en Equation 13.5.2.3 subit deux simplifications majeures :

1. Le modèle lexical $P(W|T)$ est généralement supprimé.
2. Le modèle syntaxique $P(T)$ est réduit à un modèle n-gramme.

Suppression du modèle lexical. On fait traditionnellement l'hypothèse que la probabilité d'un mot ne dépend que de sa catégorie syntaxique (Charniak 1993) :

$$P(W|T) \approx \prod_{i=1}^{m} p(w_i|t_i) \tag{13.5.2.4}$$

L'estimation de $p(w_i|t_i)$, selon la règle de Bayes, se résout comme suit :

$$p(w_i|t_i) = \frac{p(t_i|w_i)p(w_i)}{p(t_i)} \tag{13.5.2.5}$$

Si l'on fait l'hypothèse que w_i est constant quel que soit t_i, l'équation se réduit à :

$$p(w_i|t_i) = \frac{p(t_i|w_i)}{p(t_i)} \tag{13.5.2.6}$$

où $p(t_i|w_i)$ et $p(t_i)$ sont estimés par maximum de vraisemblance sur le corpus d'apprentissage :

$$p_{MV}(t_i) = \frac{c(t_i)}{\sum_{i=1}^{k} c(t_i)} \tag{13.5.2.7}$$

où $c(t_i)$ est le nombre d'occurrences de la catégorie t_i dans le corpus d'apprentissage et k est le nombre de catégories différentes.

$$p_{MV}(t_i|w_i) = \frac{c(t_i, w_i)}{c(w_i)} \tag{13.5.2.8}$$

où $c(t_i, w_i)$ représente le nombre de fois que la catégorie t_i est associée au mot w_i, et $c(w_i)$ est le nombre d'occurrences du mot w_i dans le corpus d'apprentissage. Or, dans de nombreux cas, $c(t_i, w_i)$ vaut 0. De ce fait, la plupart des systèmes ont préféré supprimer le modèle lexical du modèle de langue, réduisant ce dernier à un simple modèle syntaxique :

$$\hat{T} = \arg\max_{T} P(T) \tag{13.5.2.9}$$

Réduction du modèle syntaxique. La probabilité de la suite $P(T)$ équivaut au produit de la probabilité de chacune des catégories t_i qui la constituent. La probabilité de t_i est estimée en fonction de l'ensemble des catégories qui la précèdent dans la suite :

$$P(T) = \prod_{i=1}^{m} p(t_i|t_1, t_2, \ldots, t_{i-2}, t_{i-1}) \tag{13.5.2.10}$$

L'ensemble des catégories qui précèdent t_i dans la suite s'appelle l'*historique*. Il est cependant inconcevable d'estimer l'ensemble des historiques possibles pour chaque catégorie d'un système. Ceci demanderait un corpus gigantesque : l'ensemble des ressources disponibles n'y suffirait probablement pas. En outre, un système qui estimerait effectivement cet historique pour chaque catégorie serait inévitablement lent et inutilisable.

Pour ces deux raisons, on préfère classiquement faire l'hypothèse que la probabilité de t_i ne dépend que des $n-1$ catégories qui précèdent, et réduire l'historique à ces $n-1$ catégories (Charniak 1993). On parle de modèle n-gramme :

$$P(T) \approx \prod_{i=1}^{m} p(t_i|t_{i-n+1}^{i-1}) \tag{13.5.2.11}$$

où t_i^j note de manière concise $t_i, t_{i+1}, \ldots, t_{j-1}, t_j$. Classiquement, l'historique est réduit aux 2 catégories précédentes. Il s'agit donc d'un modèle d'ordre 3, le tri-gramme.

La probabilité du n-gramme est classiquement estimée par maximum de vraisemblance sur le corpus d'apprentissage :

$$P_{MV}(t_i|t_{i-n+1}^{i-1}) = \frac{c(t_{i-n+1}^{i})}{\sum_{t_i} c(t_{i-n+1}^{i})}) \tag{13.5.2.12}$$

où $c(t_{i-n+1}^{i})$ représente le nombre d'occurrences d'un n-gramme donné dans le corpus d'apprentissage.

Quel que soit l'ordre du modèle, certains n-grammes syntaxiques n'apparaissent que peu ou pas du tout dans le corpus d'apprentissage. Leur maximum de vraisemblance vaut dès lors 0, et toute suite comportant ces n-grammes vaudra elle-même 0, puisque les probabilités sont multipliées. Or, quelle que soit la taille du corpus d'apprentissage, il est évident que de nombreux n-grammes y seront mal représentés. Il est donc nécessaire de trouver une méthode d'estimation qui puisse pallier ce problème. On parle de méthode de *lissage*.

Traditionnellement, on distingue deux grands types de lissage :

1. Le *backoff*, où le modèle d'ordre n est remplacé par le modèle d'ordre $n-1$ lorsque le nombre d'occurrences du n-gramme vaut 0 :

$$P_{BK}(t_i|t_{i-n+1}^{i-1}) = \begin{cases} P_{MV}(t_i|t_{i-n+1}^{i-1}) & \text{si } c(t_{i-n+1}^{i}) \neq 0 \\ \lambda_{t_i|t_{i-n+1}^{i-1}} P_{BK}(t_i|t_{i-n+2}^{i-1}) & \text{sinon} \end{cases} \tag{13.5.2.13}$$

2. L'*interpolation linéaire*, où la probabilité du n-gramme est une combinaison linéaire des modèles d'ordre n à 0 :

$$P_{INT}(t_i|t_{i-n+1}^{i-1}) = \begin{array}{l} \lambda_{t_i|t_{i-n+1}^{i-1}} P_{MV}(t_i|t_{i-n+1}^{i-1}) \\ + \quad \left(1 - \lambda_{t_i|t_{i-n+1}^{i-1}}\right) P_{INT}(t_i|t_{i-n+2}^{i-1}) \end{array} \tag{13.5.2.14}$$

Dans les deux cas, le paramètre λ permet de pondérer l'importance accordée au modèle d'ordre $n-1$ dans le modèle d'ordre n. On constate en outre que λ dépend du n-gramme en cours d'estimation.

De nombreuses méthodes de lissage ont été proposées, et chacune d'elles peut être envisagée en backoff ou en interpolation linéaire.

13.5.2.3 Analyse et positionnement

Les systèmes experts peuvent certainement être conçus de manière robuste. Ils sont cependant difficilement multilingues, comme nous l'avons vu. En outre, la description de la structure syntaxique de la phrase n'est pas nécessaire aux modules qui suivent l'analyse syntaxique dans le processus de synthèse.

Sur la base de ces constats, et à l'instar d'autres systèmes actuels (Kupiec 1992, Black *et al.* 1999), notre analyse syntaxique se limite donc à un modèle de langue.

Le modèle que nous proposons veille cependant à ne pas réduire l'analyse au seul modèle syntaxique : nous réintroduisons le modèle lexical sous la forme d'un modèle d'ambiguïté lexicale.

13.5.3 Analyse syntaxique dans eLite

13.5.3.1 Introduction

Le modèle de langue que nous proposons s'inscrit dans la lignée de l'état de l'art en ce qui concerne la réduction du modèle syntaxique $P(T)$ à un n-gramme lissé. Il n'est en effet pas réaliste de considérer la totalité des historiques possibles.

Par contre, il nous semble qu'un modèle de langue, réduit à un simple modèle syntaxique, n'est pas suffisant. La conséquence de la suppression du modèle lexical $P(W|T)$ est que toute référence au mot a disparu du modèle, de sorte que deux séquences W différentes, qui proposeraient le même ensemble de suites $\{T_1, T_2, \ldots, T_{l-1}, T_l\}$, seraient traitées sans discernement et recevraient automatiquement la même analyse.

Nous avons voulu combler cette faiblesse du modèle, en réintroduisant le mot sous la forme d'un modèle d'*ambiguïté lexicale*, $P(A|T)$. Le modèle de langue que nous proposons est donc le suivant :

$$\hat{T} = \arg\max_{T} P(A|T)P(T) \tag{13.5.3.1}$$

La Section 13.5.3.2 présente l'origine et l'intérêt des classes d'ambiguïté lexicales, et le modèle d'ambiguïté lexicale que nous en avons déduit. La Section 13.5.3.3 détaille quant à elle les différentes méthodes de lissage du modèle syntaxique implémentées. Nous donnons ensuite en Section 13.5.3.4 quelques détails d'implémentation. Ce modèle, que nous avons présenté dans (Beaufort *et al.* 2002), est analysé en Section 13.5.3.5 au travers des tests réalisés et des résultats obtenus.

13.5.3.2 Définition d'un modèle lexical

Le modèle lexical que nous proposons se base sur la définition de *classe d'ambiguïté lexicale* :

Définition 13.5.1 (Classe d'ambiguïté lexicale)**.** *Une classe d'ambiguïté lexicale rassemble l'ensemble des formes lexicales qui acceptent au moins deux catégories syntaxiques et qui partagent exactement les mêmes catégories syntaxiques.*

Une classe d'ambiguïté lexicale regroupe donc toutes les formes lexicales qui partagent les mêmes catégories syntaxiques et n'en acceptent aucune autre. Voici quelques exemples de classes d'ambiguïté lexicales en français, à partir des catégories `ADJ`, `NOUN` et `VERB` :

- `{ADJ,NOUN}` : *présent*, *réel*, *riche*, *simple*, etc.
- `{ADJ,NOUN,VERB}` : *ferme*, *fous*, *pratique*, *critique*, etc.
- `{ADJ,VERB}` : *complète*, *contente*, *célèbre*, *double*, etc.
- `{NOUN,VERB}` : *angoisse*, *couvent*, *élève*, *président*, etc.

Il est important de constater que ce sont des *formes lexicales* qui appartiennent à des classes. Les différentes formes d'un même lemme peuvent donc appartenir à des classes différentes. Par exemple, le lemme *couver* recouvre entre autres les flexions *couvent* et *couvée*. Or, *couvent* appartient à la classe `{NOUN,VERB}`, tandis que *couvée* appartient à la classe `{NOUN,PARTPASSE}`.

Le concept de *classe d'ambiguïté lexicale* a été proposé par Andreewsky & Fluhr (1973). Leur analyse syntaxique opère en deux temps. Dans un premier temps, les mots d'une phrase sont remplacés par les classes d'ambiguïté auxquelles ils appartiennent. Dans un second temps, la catégorie la plus pertinente de chaque classe est retenue, sur la base de corrélations binaires et ternaires entre classes contiguës. Ces corrélations sont apprises automatiquement sur un corpus d'apprentissage. Par exemple, les termes « *la montre* » seront remplacés par leurs classes respectives, c'est-à-dire

`{DET,NOUN,PRONPERCD} {NOUN,VERBE}`

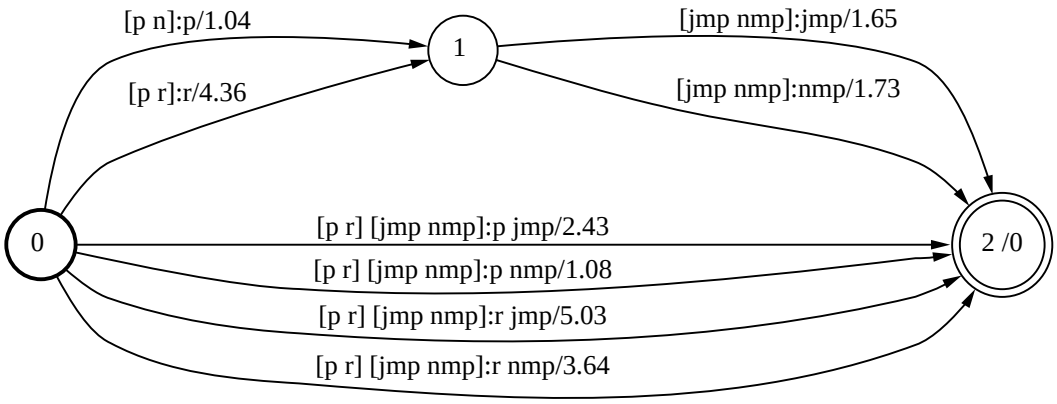


FIG. 13.6: Classes d'ambiguïté lexicales contextualisées dans un transducteur. Le choix de la catégorie, pour une classe donnée, dépend du contexte. Figure reprise de (Tzoukermann & Radev 1996)

Ensuite, la corrélation binaire

$$\texttt{\{}\overline{\texttt{DET}}\texttt{,NOUN,PRONPERCD\}}\ \textit{si}\ \texttt{\{}\overline{\texttt{NOUN}}\texttt{,VERB\}}$$

permettra de retenir la catégorie `DET` pour « *la* », pour autant que la suite de la phrase favorise le choix de `NOUN` pour « *montre* ».

Kupiec (1992) a également eu recours aux classes d'ambiguïté lexicales, afin de réduire le nombre des paramètres d'apprentissage de son analyseur HMM. Tzoukermann & Radev (1996) et Kempe (2000) les ont ensuite employées comme catégories de leurs transducteurs syntaxiques, en lieu et place des mots ; en quelque sorte, ils réunissent le modèle syntaxique et le modèle lexical au sein d'un seul modèle (cf. Figure 13.6).

Notre approche est différente. Nous conservons une distinction entre le modèle syntaxique et le modèle lexical : les classes d'ambiguïté n'interviennent que dans le modèle lexical, et sont entraînées séparément du modèle syntaxique.

Pondération des classes. Nous attribuons à chaque catégorie d'une classe donnée une probabilité d'émission, propre à la classe et déterminée à partir du corpus d'apprentissage. Pour une classe c_i, la probabilité d'émission d'une catégorie t_j donnée est déterminée en normalisant la somme des occurrences de cette catégorie pour chaque mot w_k de la classe c_i présent dans le corpus d'entraînement :

$$p(t_j|c_i) = \frac{\sum_{w_k \in c_i} c(w_k, t_j)}{\sum_{t_j} \sum_{w_k \in c_i} c(w_k, t_j)} \qquad (13.5.3.2)$$

La Table 13.11 donne un exemple de pondération des catégories d'une classe d'ambiguïté.

Mot	ADJ	NOUN	PARTPASSE	Total
mort	2	8	3	13
passé	1	1	8	10
venu	3	1	11	15
Total	6	10	22	**38**
Probabilité	0,1579	0,2632	0,5789	**1**

TAB. 13.11: Pondération d'une classe d'ambiguïté lexicale. La classe illustrée est {ADJ,NOUN,PARTPASSE}

Modèle d'ambiguïté lexicale. En définissant les classes d'ambiguïté lexicales, notre idée initiale était de les utiliser exclusivement à la place du modèle lexical. La suite de mots W serait dans ce cas remplacée par la suite de classes correspondantes C :

$$P(W|T) \approx P(C|T) \approx \prod_{i=1}^{m} p(c_i|t_i) \tag{13.5.3.3}$$

Or, le modèle lexical ne peut se limiter aux classes d'ambiguïté lexicales :

1. Il serait dommage de recourir à la classe d'ambiguïté lorsque le mot appartient lui-même au corpus d'entraînement. En effet, comme l'illustre la Table 13.11, la distribution des catégories d'une classe entre les mots de cette classe varie fortement. On constatera par exemple que *mort* est plus probable en tant que NOUN, alors que *venu* est plus probable en tant que PARTPASSE.
2. Il est possible que ni le mot, ni sa classe n'aient été observés sur le corpus d'entraînement. Dans ce cas, il n'y a pas de raison de différencier la probabilité de ses catégories.
3. Le modèle doit également tenir compte des mots qui ne présentent qu'une seule catégorie.

Afin de tenir compte de ces différents cas de figure, nous proposons de substituer un modèle d'*ambiguïté lexicale* au modèle lexical. Nous remplaçons donc la suite de mots W par la suite d'ambiguïtés lexicales A :

$$P(W|T) \approx P(A|T) \approx \prod_{i=1}^{m} p(a_i|t_i) \tag{13.5.3.4}$$

où a_i est défini comme suit :

$$p(a_i|t_i) = \begin{cases} p(w_i|t_i) & \text{si } w_i \text{ a été observé sur le corpus,} \\ p(c_i|t_i) & \text{si } w_i \in c_i \text{ et } c_i \text{ a été observé sur le corpus,} \\ \frac{1}{N} & \text{sinon} \end{cases} \tag{13.5.3.5}$$

où N est le nombre de catégories acceptées par w_i.

En combinant le modèle d'ambiguïté lexicale au modèle syntaxique, nous obtenons le modèle de langue tel que nous l'avons défini en Equation 13.5.3.1. Le mot est donc réintroduit dans l'estimation syntaxique.

13.5.3.3 Lissage du modèle n-gramme

Dans leur revue de l'ensemble des méthodes de lissage existantes, Chen & Goodman (1998) ont montré que quatre d'entre elles se dégagent particulièrement :

1. Jelinek-Mercer (Jelinek & Mercer 1980).
2. Witten-Bell (Witten & Bell 1991).
3. Absolute Discounting (Ney *et al.* 1994).
4. Kneser-Ney (Kneser & Ney 1995).

Ces méthodes diffèrent sur deux points :

1. Elles proposent des méthodes différentes pour calculer les facteurs λ.
2. Certaines n'utilisent pas P_{MV} dans le calcul des modèles inférieurs à l'ordre n, mais une autre estimation.

Chen & Goodman (1998) ont en outre démontré que, quelle que soit la méthode de lissage, l'interpolation linéaire donne de meilleurs résultats que le backoff. Nous avons de ce fait implémenté les quatre méthodes dans leur version interpolée. Afin d'estimer les résultats de ces approches par rapport à ceux obtenus avec un système de base, nous avons également implémenté un lissage additif (Lidstone 1920). Nous ne donnons ici qu'une description sommaire des différentes méthodes de lissage que nous avons testées, et invitons le lecteur intéressé à se reporter aux articles mentionnés.

Lissage additif. Le lissage additif ne se conçoit ni en version backoff, ni en version interpolée. Pour éviter les comptages nuls, la méthode considère que chaque n-gramme apparaît δ fois plus que le nombre d'occurrences présentes dans les données d'entraînement, avec $0 < \delta \leq 1$:

$$P_{ADD}(t_i|t_{i-n+1}^{i-1}) = \frac{\delta + c(t_{i-n+1}^{i})}{\delta|V| + \sum_{t_i} c(t_{i-n+1}^{i})} \tag{13.5.3.6}$$

où $|V|$ représente le nombre de mots dans le corpus d'entraînement. Dans notre système, nous avons fixé le facteur δ à 1.

Lissage Jelinek-Mercer. Cette méthode implémente la forme générale de l'interpolation linéaire. Les facteurs λ sont calculés à l'aide de l'algorithme de Baum-Welch (Baum 1972), à partir de données extérieures au corpus d'apprentissage.

La taille réduite de notre corpus d'apprentissage ne nous a pas permis d'en réserver une partie à l'entraînement seul des facteurs λ. Nous désirions néanmoins conserver ce lissage. Dans une simplification extrême, de nombreux tests nous ont permis de fixer un seul $\lambda = 0.9$ pour les modèles d'ordre 1 à 3.

Lissage Witten-Bell. Il réduit λ au nombre de mots uniques qui suivent l'historique t_{i-n+1}^{i-1} :

$$\lambda = N_{1+}(t_{i-n+1}^{i-1}\bullet) = |\{t_i : c(t_i|t_{i-n+1}^{i-1}t_i) > 0\}| \tag{13.5.3.7}$$

où N_{1+} désigne le nombre de n-grammes qui ont une ou plusieurs occurrences dans le corpus d'entraînement, et $\bullet$ la variable (t_i) sur laquelle on somme.

Lissage Absolute Discounting. La probabilité du n-gramme d'ordre n n'est plus calculée en multipliant le maximum de vraisemblance par un facteur λ, mais en ôtant un décompte fixe $0 < D \leq 1$ au numérateur du maximum de vraisemblance :

$$P_{ABS}(t_i|t_{i-n+1}^{i-1}) = \frac{\max\{c(t_{i-n+1}^{i}) - D, 0\}}{\sum_{t_i} c(t_{i-n+1}^{i})} \tag{13.5.3.8}$$

En outre, pour que la distribution totale somme à 1, le facteur $(1 - \lambda)$ est également affecté par ce décompte :

$$1 - \lambda_{t_i|t_{i-n+1}^{i-1}} = \frac{D}{\sum_{t_i} c(t_{i-n+1}^{i})} N_{1+}(t_{i-n+1}^{i-1}\bullet) \tag{13.5.3.9}$$

D est calculé de la manière suivante :

$$D = \frac{n_1}{n_1 + n_2} \tag{13.5.3.10}$$

où n_1 et n_2 sont le nombre total de n-grammes qui ont, dans la base d'entraînement, respectivement une et deux occurrences, et où n est l'ordre du modèle en cours d'interpolation.

Lissage Kneser-Ney. Largement inspiré du précédent, l'idée maîtresse de cette méthode est cependant que l'interpolation d'un $(n-i)$-gramme ne doit pas être liée à ses occurrences, mais au nombre de catégories différentes que ce $(n-i)$-gramme suit. Tout modèle d'ordre inférieur à n est calculé de la manière suivante :

$$P_{KN}(t_i|t_{i-n+2}^{i-1}) = \frac{N_{1+}(\bullet t_{i-n+2}^{i})}{\sum_{t_i} N_{1+}(\bullet t_{i-n+2}^{i})} \tag{13.5.3.11}$$

En reconnaissance de la parole, les résultats de Kneser-Ney dépassent largement ceux obtenus par les autres lissages présentés (Chen & Goodman 1998).

13.5.3.4 Implémentation

Logarithmes. Pour les besoins de l'implémentation, les produits de probabilités ont été remplacés par des sommes de logarithmes négatifs. De ce fait, la meilleure suite $\hat{T}$ est celle de poids minimum :

$$\hat{T} = \arg\min_{T} LOG(A|T) + LOG_{INT}(T) \tag{13.5.3.12}$$

$$LOG(A|T) = \sum_{i=1}^{m} -\log\, p(a_i|t_i) \tag{13.5.3.13}$$

$$LOG_{INT}(T) = \sum_{i=1}^{m} -\log\, p_{INT}(t_i|t_{i-n+1}^{i-1}) \tag{13.5.3.14}$$

Programmation dynamique. Nous avons jusqu'à présent considéré les combinaisons possibles des ensembles de catégories $\{t^1, \ldots, t^k\}$ attribuables à chaque mot w_i de la suite W comme l'ensemble des suites $\{T_1, T_2, \ldots, T_{l-1}, T_l\}$. Cet ensemble de suites peut également être considéré comme un treillis de solutions.

La Figure 13.7 donne un exemple de treillis syntaxique. Dans le treillis, chaque nœud correspondant à un mot est connecté à l'ensemble des nœuds correspondant aux mots qui l'entourent. Dans l'exemple que nous donnons, on constate que la phrase commence par le mot-clef `<BOS>`, qui indique le début de la phrase [17], et termine par le mot-clef `<EOS>`, qui en indique la fin [18]. Ceci permet d'estimer la probabilité d'un n-gramme en début et en fin de phrase. Chaque mot-clef est associé à une catégorie qui lui est propre.

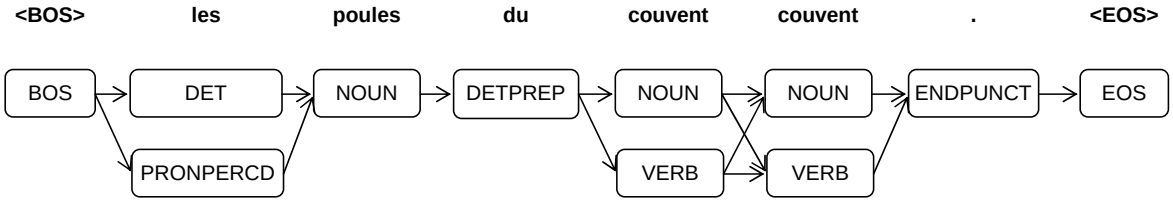


FIG. 13.7: Treillis de catégories syntaxiques

L'intérêt du treillis est qu'il permet de rechercher la suite de catégories syntaxiques qui maximise le modèle de langue (cf. Equation 13.5.3.1) par programmation dynamique. La programmation dynamique (Bellman & Dreyfus 1962) est une approche qui permet de rapidement trouver le meilleur chemin entre un nœud initial I et un nœud final F dans un treillis de possibilités, en évitant de calculer le coût de tous les chemins du treillis entre I

[17] BOS : *Begin Of Sentence.*
[18] EOS : *End Of Sentence.*

et F. L'idée est de considérer la solution globale comme la suite des meilleures hypothèses locales. Dans ce modèle, le coût d'un nœud i du treillis depuis le nœud I vaut :

$$\texttt{C}(i, I) = w(i) + \max_{h=1}^{n}\{d(i, h) + \texttt{C}(h, I)\} \tag{13.5.3.15}$$

où $w(i)$ note le poids du nœud i, h est un prédécesseur de i, n est le nombre de prédécesseurs de i, et $d(i, h)$ représente le coût de transition entre i et h. La solution globale correspond à $\texttt{C}(F, I)$, et peut être construite pour autant que l'on ait conservé, dans chaque nœud, le meilleur de ses prédécesseurs.

Tel que décrite, la programmation dynamique peut uniquement gérer un modèle de langue d'ordre 2 (bi-gramme). Or, notre analyse doit gérer un tri-gramme. Ceci signifie qu'un nœud du treillis doit explicitement ou implicitement tenir compte de ses deux prédécesseurs.

Nous avons choisi d'utiliser une représentation implicite du tri-gramme. La Figure 13.8 illustre le principe. La structure est un vecteur de pointeurs dont chaque position

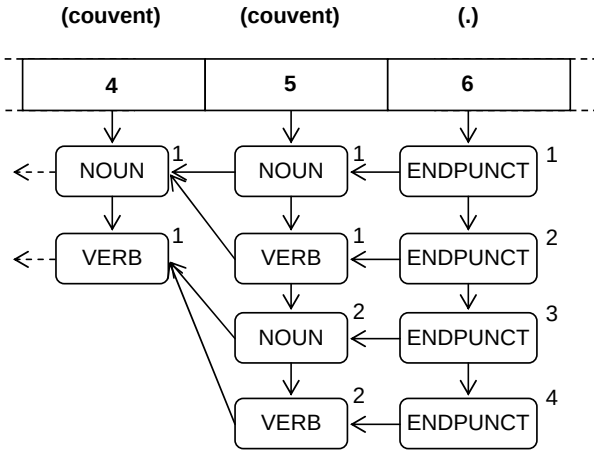


FIG. 13.8: Programmation dynamique : structure de données

correspond à une unité linguistique. Une position pointe vers le premier de ses nœuds. Chaque nœud pointe quant à lui vers son unique prédécesseur et son premier frère. La représentation du tri-gramme est implicite dans le sens où, pour une position donnée, la même catégorie de l'unité linguistique peut être représentée par plusieurs nœuds, chacun d'eux pointant vers un prédécesseur différent.

Par exemple, si le meilleur chemin, en position 6, passe par le nœud `ENDPUNCT` numéro 2, ce nœud pointe obligatoirement vers le nœud `VERB` numéro 1 de la position 5, qui implique le nœud `NOUN` de la position 4. Ceci signifie que, de tous les tri-grammes possibles, celui qui a reçu la meilleure estimation en position 6 est `NOUN VERB ENDPUNCT`. Cette estimation est quant à elle la somme du modèle lexical $p(.|\texttt{ENDPUNCT})$, du modèle syntaxique $p_{INT}(\texttt{ENDPUNCT}|\texttt{NOUN,VERB})$ et du poids du prédécesseur, le nœud `VERB` numéro 1 de la position 5.

Lorsque tous les nœuds ont été construits, la recherche du meilleur chemin est simplifiée par cette structure. Le principe est de rechercher le nœud qui présente le meilleur poids sur la dernière position du vecteur. Lorsque ce nœud est trouvé, l'algorithme a uniquement à parcourir les prédécesseurs de ce nœud jusqu'à la position initiale du vecteur. Cette représentation optimise donc le temps de traitement, au détriment de la place, mais dans une mesure tout à fait négligeable au vu de la légèreté de la structure utilisée.

13.5.3.5 Tests et résultats

Méthodologie et données. L'objectif principal de nos tests était d'évaluer l'apport que constitue le modèle d'ambiguïté lexicale. En second lieu, nous voulions également établir une hiérarchie des lissages implémentés. Nous avons donc testé deux versions de notre analyseur : la première réduit l'analyse au modèle syntaxique, la seconde intègre le modèle d'ambiguïté lexicale. Dans les deux cas, les 5 méthodes de lissage présentées ont été utilisées, ainsi qu'une version non lissée de notre analyseur.

Notre corpus, « Le mot et l'idée », compte 66 619 mots. Il a été constitué par l'AUPELF-UREF [19] (maintenant l'AUF [20]) dans le cadre de l'action de recherche concertée « Synthèse vocale ». Ce corpus a été étiqueté automatiquement à l'aide d'un système par règles, Vertex (Mertens 2001), et corrigé manuellement.

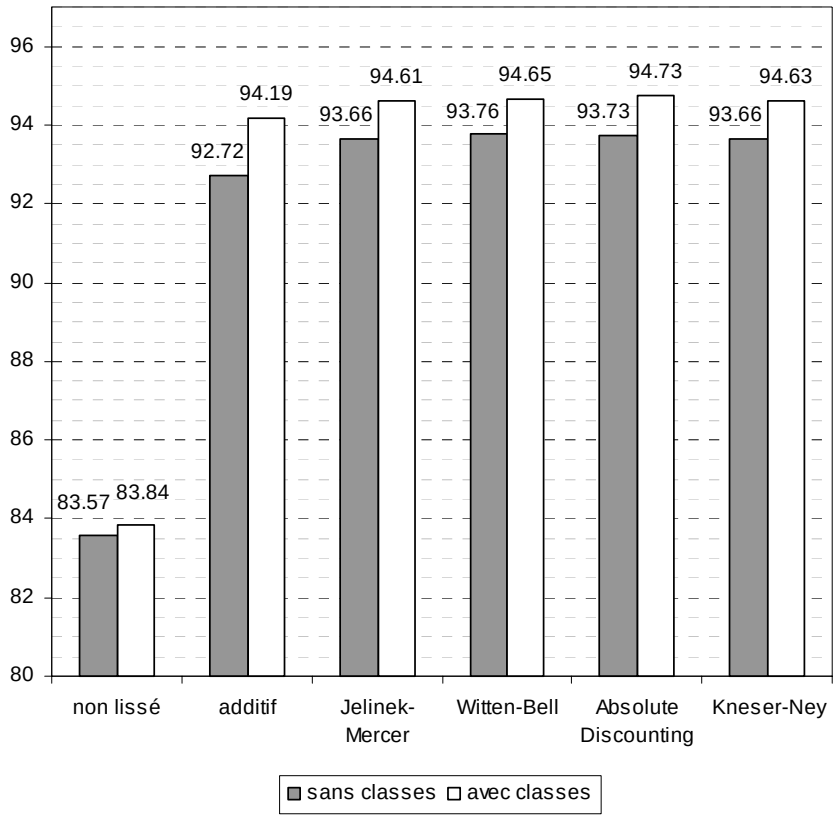
Le test a été réalisé par *n-fold-cross-validation* (Kohavi 1995). Le principe est de diviser le corpus en n parties égales et d'utiliser chaque partie 1 fois comme corpus de test et $n-1$ fois dans le corpus d'entraînement. L'intérêt de cette procédure est qu'elle permet d'exécuter n fois un algorithme sur des données extérieures à l'entraînement. En l'occurrence, nous avons coupé le corpus en 10 parties.

Résultats et analyse. Sur nos dix tests, nous avons obtenu en moyenne 95 classes d'ambiguïté, contenant au plus 4 catégories syntaxiques. La Table 13.12 présente la moyenne des résultats des 6 méthodes sur les 10 tests. La Table 13.13 indique le nombre d'erreurs propres à chaque méthode.

D'emblée, on constate toute l'importance d'un lissage quel qu'il soit, puisque la méthode non lissée est à 10% en-dessous du lissage le moins performant qui, sans surprise, est le lissage additif.

Sans classes – avec classes. Version non lissée mise à part, l'amélioration apportée par le modèle d'ambiguïté (cf. Table 13.12) est en moyenne de 1.06% (707 mots). Ceci dit, le lissage additif profite d'une amélioration nettement meilleure (1.47%) que les méthodes d'interpolation (0.95%). Ceci semble montrer que l'interpolation et le modèle d'ambiguïté convergent vers la même analyse dans un certain nombre de cas.

[19] AUPELF-UREF : Agence Francophone pour l'Enseignement Supérieur et la Recherche.
[20] AUF : Agence Universitaire de la Francophonie.



TAB. 13.12: Moyenne (en %) d'étiquetage correct sur les dix corpus-tests, avec et sans modèle d'ambiguïté lexicale

	sans classes		avec classes	
	erreurs propres	%	erreurs propres	%
Non lissé	6974	63.86	7266	67.62
Additif	574	11.91	474	12.31
Jelinek-Mercer	15	0.36	18	0.50
Witten-Bell	3	0.07	19	0.53
Abs. Discount.	4	0.10	3	0.09
Kneser-Ney	19	0.45	51	1.42

TAB. 13.13: Total des erreurs propres sur les dix tests

Il faut noter que les performances de l'analyseur non lissé ne sont que très légèrement améliorées par la présence du modèle d'ambiguïté (0.27%, soit 180 mots). Ceci est évidemment dû aux n-grammes de probabilité nulle, qui rendent l'analyse de certaines phrases complètement impossible. Dans ce cas, c'est la catégorie qui apparaît *par hasard* en tête de liste qui est choisie.

Au sein de l'interpolation. Les 4 méthodes ont des résultats fort similaires (cf. Table 13.12), que ce soit avec ou sans modèle d'ambiguïté lexicale. Contrairement à ce qui a été constaté en reconnaissance de la parole, Kneser-Ney ne surpasse pas les résultats des autres lissages, et compte le plus d'erreurs propres (cf. Table 13.13), le phénomène étant encore amplifié en présence du modèle d'ambiguïté. Ceci ne signifie pas pour autant que ce lissage soit non-performant, mais le principe d'interpoler un $(n-i)$-gramme, en fonction du nombre de termes différents qu'il suit, impose probablement d'utiliser un corpus comptant au minimum plusieurs centaines de milliers de mots. On constate d'ailleurs que le corpus utilisé en reconnaissance (Chen & Goodman 1998) dépassait le million de mots, alors que le nôtre n'en compte même pas 100 000.

On notera cependant que les erreurs propres de Kneser-Ney présentent très peu d'homographes hétérophones, au contraire des autres méthodes. Les erreurs de Kneser-Ney sont par contre beaucoup plus fréquentes sur les paires de catégories dont le comportement syntaxique est fort similaire. Par exemple, Kneser-Ney gère mal la paire {`ADJ,PARTPASSE`} derrière le verbe copule *être* : ainsi, la méthode va étiqueter `ADJ` la forme *venu* dans *il est venu*, probablement du fait d'autres phrases fort proches, comme *il est gentil* ou *il est blanc*. Dans le cadre de la synthèse de la parole, ces erreurs sont évidemment nettement moins gênantes que celles commises sur les homographes hétérophones.

Au terme de ces tests, Kneser-Ney ne donne pas les meilleurs résultats en analyse syntaxique, mais son principe d'interpolation distingue cette méthode des autres et offre de meilleurs résultats sur les erreurs qui comptent en synthèse : celles qui sont audibles. Le modèle d'ambiguïté lexicale, quant à lui, améliore significativement les résultats.

13.5.3.6 Conclusion

Nous avons implémenté un analyseur syntaxique probabiliste. Comme les systèmes actuels, le modèle de langue est un n-gramme lissé, mais contrairement à eux, le modèle lexical n'est pas supprimé de l'estimation.

Nous avons testé quatre méthodes de lissage par interpolation : Jelinek-Mercer, Witten-Bell, Absolute Discounting et Kneser-Ney. Leurs résultats sont fortement similaires. Kneser-Ney, qui surpasse les autres méthodes en reconnaissance, affiche ici des performances équivalentes, probablement du fait de la petite taille de notre corpus d'apprentissage. Par contre, cette méthode analyse mieux les homographes hétérophones, ce qui contribue incontestablement à la qualité globale du système de synthèse.

Nous avons réintroduit le modèle lexical sous la forme d'un modèle d'ambiguïté

lexicale, basé lui-même sur la notion de classes d'ambiguïté lexicales. Nos résultats montrent que ce modèle améliore significativement l'analyse syntaxique. Il n'est donc pas inutile de considérer le mot dans l'estimation d'un modèle de langue.

13.6 Le point sur l'analyse linguistique présentée

L'analyse linguistique d'eLite se fonde sur le concept d'*unité linguistique*, qui simplifie l'analyse syntaxique en favorisant une approche de plus haut niveau, et permet aux différents modules de spécialiser les traitements en fonction de l'unité.

Outre le traitement approprié des différents tokens détectés par le pré-processeur (URIs, dates, mesures, etc.), l'analyseur morphologique produit principalement une analyse flexionnelle des formes lexicales, et réalise la recomposition des mots composés. Ce module s'appuie sur des lexiques qui le rendent indépendant de la langue, pour autant que celle-ci soit flexionnelle.

L'analyseur syntaxique est un modèle de langue, dont le modèle syntaxique est lissé par interpolation linéaire, et dont le modèle lexical a été conservé sous la forme d'un modèle d'ambiguïté lexicale. Ce modèle d'ambiguïté lexicale est l'élément qui apporte le plus aux performances du système. Le modèle est entraîné sur un corpus contenant simplement des phrases où chaque mot est associé à sa catégorie syntaxique. Le modèle est donc facilement portable, puisqu'il est assez aisé de trouver un corpus étiqueté pour de nombreuses langues. Ce modèle est en outre robuste, étant donné qu'il produit systématiquement une analyse, quelle que soit la phrase. En outre, l'analyse produite est fiable : les tests que nous avons réalisés en français situent le taux d'erreurs autour des 5,5%.

L'analyse linguistique dans son ensemble est robuste et efficace. Elle n'est cependant pas optimale :

1. L'analyse ne s'applique qu'aux langues flexionnelles.
2. La division de l'analyse morpho-syntaxique en deux modules impose de multiples accès à la DLS : l'analyseur morphologique crée de multiples analyses dans la DLS, dont la plupart seront ensuite supprimées par l'analyseur syntaxique.
3. Au sein de l'analyseur morphologique, l'analyse flexionnelle de toutes les formes lexicales est certainement inutile, étant donné que nous ne gardons de cette analyse que les formes qui proviennent d'un lemme connu de nos lexiques. Les mots inconnus ne profitent pas de cette analyse, parce que le module ne fournit aucun moyen de *quantifier* l'importance à accorder aux différentes analyses retournées par le trie.

Cette division en deux modules complique en outre la mise en place de la correction orthographique telle que nous l'avons envisagée (cf. Postulat 15.1.2 et Hypothèse 15.2.2). Ce point sera cependant abordé au Chapitre 15, étant donné que sa justification demande de poser préalablement l'état de l'art en correction orthographique.

Chapitre 14

Etat de l'art en correction

Dans ce chapitre, nous commençons par présenter une typologie succincte des erreurs susceptibles de survenir dans un texte. Nous donnons ensuite un aperçu des niveaux de complexité en correction orthographique.

Sur cette base, nous abordons le problème de la détection des erreurs qui résultent en des non-mots, avant de détailler les méthodes de correction qui les concernent. Nous proposons ensuite un rapide survol des nombreuses méthodes dédiées à la correction des erreurs dépendantes du contexte.

Ce chapitre se referme sur une rappel des différents points abordés et sur une estimation de ce qui peut s'appliquer à la correction dans le cadre de la synthèse de la parole.

Cet état de l'art est fortement inspiré de l'excellente synthèse du domaine réalisée par Kukich (1992b), que nous avons complétée de quelques articles plus récents (Gale *et al.* 1994, Littlestone & Warmuth 1994, Valiant 2004, Yarowsky 1994, Brill 1995, Golding 1995, Valiant 1995, Golding & Schabes 1996, Golding & Roth 1996, Jones & Martin 1997, Mangu & Brill 1997, Golding & Roth 1999, Mohri 1996, Brill & Moore 2000, Volk 2001, Zhu & Rosenfeld 2001, Toutanova & Moore 2002, Keller & Lapata 2003, Kilgarriff & Grefenstette 2003, Strohmaier *et al.* 2003, Baroni & Bernardini 2004, Cucerzan & Brill 2004, Hirst & Budanitsky 2005, Williams & Zobel 2005, Lazarov 2006, Ringlstetter *et al.* 2007, Schaback & Li 2007).

14.1 Typologie des erreurs

14.1.1 Types d'erreurs

Les erreurs présentes dans un texte se répartissent entre quatre types distincts :

1. Les non-mots.
2. les erreurs syntaxiques.
3. Les erreurs sémantiques ou pragmatiques.

4. Le non-respect des conventions.

Les non-mots. Ce que l'on appelle un *non-mot* est un mot inconnu du dictionnaire utilisé par le système de correction. La notion anglaise de *Out-Of-Vocabulary words* (OOV) exprime donc mieux ce qu'est le non-mot :

- Un mot réellement inexistant (*élèbe* pour *élève*, par exemple).
- Un mot absent du dictionnaire parce que celui-ci est incomplet.
- Un mot emprunté à une langue étrangère (*tuning*) ou créé à partir d'une racine étrangère (*dump-er*).

Par opposition à OOV, un mot qui appartient au dictionnaire est qualifié de *In-Vocabulary words* (IV). Nous employons ces acronymes dans les pages qui suivent.

Les erreurs syntaxiques. Il s'agit de mots existants, mais syntaxiquement incorrects. Par exemple :

- *Les personnes qu'il a **vu** hier sont là.* (l'accord au féminin pluriel manque).
- *Il **dois** apprendre.* (le sujet est une troisième personne du singulier).
- *Je **serais** à la gare à 9h00.* (confusion futur/conditionnel).
- *La police a **arrêter** un individu.* (confusion participe passé/infinitif).

Les erreurs sémantiques ou pragmatiques. Il s'agit de mots existants, mais sémantiquement ou pragmatiquement inadaptés. Par exemple :

- sémantique : *sceptique* ↔ *septique*, *tort* ↔ *tord*, *palais* ↔ *palet*, *les eaux* ↔ *les os*, *pallier* ↔ *palier*, etc.
- pragmatique : *Louis XIV mourut en 1815* (1715).

Le non-respect des conventions. Il s'agit des conventions liées à un type particulier d'écrit. Par exemple, dans une lettre, l'entête ne peut pas contenir le nom du destinataire (*Monsieur Dupont, ...*).

Notons dès à présent que nous ne nous intéressons pas à ce type d'« erreurs », étant donné que la correction orthographique en synthèse de la parole a pour objectif de supprimer les erreurs audibles, mais sans modifier le message tel qu'il a été pensé par l'utilisateur (cf. Section 15.2.1).

14.1.2 Causes des erreurs

Si le non-respect des conventions est exclusivement dû à l'utilisateur qui ne connaît pas ou ne respecte pas les conventions, les trois autres types d'erreurs partagent des origines communes : elles ressortissent soit aux erreurs d'édition, soit aux erreurs linguistiques, et sont fonction de la longueur du mot.

14.1.2.1 Les erreurs d'édition

Ces erreurs ne sont pas dues aux connaissances de l'utilisateur, mais aux limites imposées par le média utilisé. Les médias concernés ici sont le clavier et les systèmes de reconnaissance de caractères [1]. Les erreurs générées varient selon le média.

Erreurs typographiques. Une erreur typographique est le résultat de la proximité des touches sur le clavier et de la maîtrise parfois approximative du clavier par l'utilisateur. Quatre erreurs typographiques sont possibles : la substitution d'une lettre pour une autre, l'insertion ou la suppression d'une lettre, ou la transposition de deux lettres (cf. Table 14.1).

Substitution	*élève* → *élèbe*, *ferme* → *germe*, *patte* → *parte*
Insertion	*élève* → *élèbve*, *par* → *part*, *parles* → *par les*
Suppression	*élève* → *élèe* *part* → *par*, *sceptique* → *septique*, *par les* → *parles*
Inversion	*élève* → *élèev*, *ne* → *en*, *un* → *nu*

Tab. 14.1: Erreurs typographiques

Quoi qu'il en soit, on constate que l'erreur peut résulter en un OOV (*élèbe*/*élève*), une erreur syntaxique (*par*/*part*), une erreur sémantique (*septique*/*sceptique*) ou une erreur de frontière (*par les*/*parles*).

On constate de manière générale que les erreurs typographiques sont plus fréquentes entre caractères proches qu'entre caractères éloignés sur le clavier. La fréquence des erreurs varie cependant fortement en fonction du type de clavier (*qwerty*, *azerty*, etc.) et de la position des caractères sur le clavier. Les études qui ont été réalisées montrent en outre que les erreurs commises sur la première lettre du mot sont assez rares. Selon les études cependant, le taux d'erreurs sur la première lettre varie de 1,5% à 15% (Yannakoudakis & Fawthrop 1983, Pollock & Zamora 1983, Mitton 1987, Kukich 1992a).

Erreurs de reconnaissance. Les erreurs de reconnaissance ont deux causes. La première est la similitude de forme entre caractères ou entre groupes de caractères. La seconde est la

[1] Les systèmes de reconnaissance de caractères sont utilisés lors de la dernière étape du processus de numérisation de documents scannés.

Substitution 1/1	*hier* → *hler*, *parte* → *patte*
Substitution $n/1$ et $1/n$	*dame* → *darne*, *garni* → *gami*
Suppression d'espaces	*il se* → *ilse*, *par les* → *parles*
Insertion d'espaces	*parles* → *par les*, *lame* → *lam e*
Suppression de symboles (ponctuations et diacritiques)	. , - ' ...

TAB. 14.2: Erreurs de reconnaissance

qualité de la numérisation, qui peut fortement dégrader la forme des caractères proposés au système de reconnaissance. De ce fait, une erreur de reconnaissance peut être une substitution 1/1, $1/n$ ou $n/1$, la suppression ou l'insertion d'espaces, ou la suppression de petits symboles difficiles à repérer (cf. Table 14.2).

L'erreur peut résulter en un OOV (*hler*/*hier*), une erreur sémantique (*darne*/*dame*) ou une erreur de frontière (*par les*/*parles*).

14.1.2.2 Les erreurs linguistiques

Ces erreurs sont le fait de la maîtrise approximative de la langue par l'utilisateur. On distingue classiquement les erreurs lexicales des erreurs syntaxiques.

Erreurs lexicales. Ces erreurs sont dues à la méconnaissance de l'orthographe d'un mot ou à la non-compréhension du sens du message. Dans le premier cas, on parle d'*erreur phonétique* et dans le second, d'*erreur cognitive*. Il est cependant souvent délicat de faire la distinction entre les deux.

Ces erreurs peuvent donner des OOVs ({*bato*, *batau*} pour *bateau*) ou des erreurs sémantiques (*sceptique* ↔ *septique*) dont certaines comportent des erreurs de frontière (*l'attention* ↔ *la tension*, *la mie* ↔ *l'amie* [2]).

Erreurs syntaxiques. Ces erreurs dénotent le manque de maîtrise par l'utilisateur des règles qui président à l'agencement des mots de la phrase. Elles résultent toujours en mots existants, mais sont parfois difficiles à distinguer des erreurs typographiques (« *les gens qu'il a vu* » peut être le résultat d'une simple suppression).

[2]Notons que ce genre de phénomènes jalonne l'histoire de la langue. Par exemple, « *mon amie* » se disait « *m'amie* » en ancien français et s'écrivait « *mamie* » dans les manuscrits. L'origine du mot a été mal interprétée en moyen français, ce qui a donné l'expression « *ma mie* ».

14.1.2.3 Influence de la longueur du mot

Des études montrent que l'écart entre une forme erronée et la forme correcte correspondante est rarement supérieur à deux caractères (Peterson 1986, Yannakoudakis & Fawthrop 1983). Les erreurs effectuées sur les mots courts sont en outre plus rares (aux alentours de 1,5%), mais plus difficiles à corriger, en partie parce que l'information interne au mot est fonction de sa longueur, ce qui implique qu'un mot court présente moins d'information interne qu'un mot long .

14.2 Niveaux de complexité en correction orthographique

Trois niveaux de difficultés sont à distinguer en correction orthographique :

1. Mots isolés *vs* Mots en contexte.
2. Détection *vs* Correction.
3. Correction interactive *vs* Correction automatique.

14.2.1 Mots isolés *vs* Mots en contexte

La notion de *mot isolé* sous-entend qu'aucune information externe au mot n'intervient dans le traitement. Un « simple » dictionnaire suffit donc, mais seuls les OOVs peuvent être traités par le système. Le traitement des mots en contexte implique de disposer, outre un dictionnaire, d'informations contextuelles, de niveau syntaxique, sémantique et/ou pragmatique, qui permettront d'évaluer la pertinence de différentes solutions en fonction du contexte dans lequel le mot à traiter se trouve.

14.2.2 Détection *vs* Correction

S'il y a correction, celle-ci suit obligatoirement une phase de détection de l'erreur. On peut donc à juste titre considérer que la correction ajoute une difficulté supplémentaire à la simple détection, puisqu'elle doit *constituer* une liste de candidats et *choisir* le meilleur d'entre eux.

14.2.3 Correction interactive *vs* Correction automatique

La correction interactive, ou *correction supervisée*, ne choisit pas elle-même la correction à apporter, mais propose à l'utilisateur une liste de candidats possibles, et laisse à celui-ci le soin de choisir la correction qui lui convient. Ce comportement est typiquement celui des traitements de texte. La correction automatique, quand à elle, détermine elle-même si la correction est opportune ou non, et choisit la solution à retenir. Ce mode de correction est utile lorsqu'aucune interaction n'est possible avec l'utilisateur. C'est le cas, par exemple, en synthèse de la parole. La correction automatique est évidemment celle qui, de nos jours,

devient la plus demandée, mais aucun système automatique ne fournit actuellement les résultats espérés.

14.2.4 Synthèse

Plusieurs de ces difficultés peuvent être prises en compte au sein d'un même correcteur orthographique, ce qui déterminera le niveau de complexité du correcteur, qui peut aller de la simple détection des OOVs à la correction automatique de mots isolés et en contexte.

Notons que le bien-fondé de la correction a lui-même été remis en question par certains chercheurs (Bentley 1985), au vu de la productivité de la langue et de la vitesse à laquelle certains mots entrent dans le lexique et en sortent.

Les systèmes existants se répartissent entre les trois domaines suivants :

1. Détection des OOVs
2. Correction des OOVs
3. Correction des erreurs sur IVs

14.3 Détection des OOVs

Les deux techniques majeures qui ont été approfondies en détection des OOVs sont la recherche dans **les dictionnaires** et **l'analyse par modèles n-gramme**.

14.3.1 Les dictionnaires

Les systèmes basés sur un dictionnaire vérifient simplement la présence de chaque terme dans le dictionnaire. Si un mot en est absent, il est étiqueté comme un OOV.

La recherche dans un dictionnaire est simple. Cependant, le temps de réponse du système croît rapidement à partir du moment où la taille du dictionnaire dépasse quelques centaines de mots. Pour résoudre ce problème, les dictionnaires ont été représentés sous la forme de tables de hashage, ou de machines à états finis (cf. Figures 14.1 et 14.2) (Knuth 1973, Turba 1981, Aho 1990, Mohri 1996).

Pour limiter la place mémoire utilisée, certains systèmes ont évité de représenter la totalité des mots de la langue dans le dictionnaire, ne conservant que les formes canoniques des mots. Les flexions, préfixes et suffixes sont dans ce cas préalablement enlevés de la forme à rechercher dans le dictionnaire (Peterson 1980, Elliott 1988).

Avantage. L'accès au contenu du dictionnaire est très rapide lorsque celui-ci est représenté sous la forme d'une table de hashage ou d'une machine à états finis. Le temps de recherche est linéairement proportionnel à la taille du mot à rechercher : on est en $O(n)$ pour un mot de longueur n.

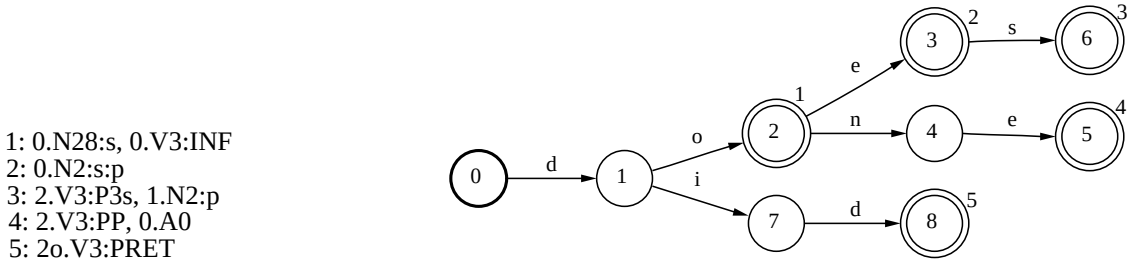


FIG. 14.1: Le dictionnaire sous la forme d'un FSA. Figure reprise de (Mohri 1996). Les chiffres en indice correspondent à des codes d'analyse morphologique

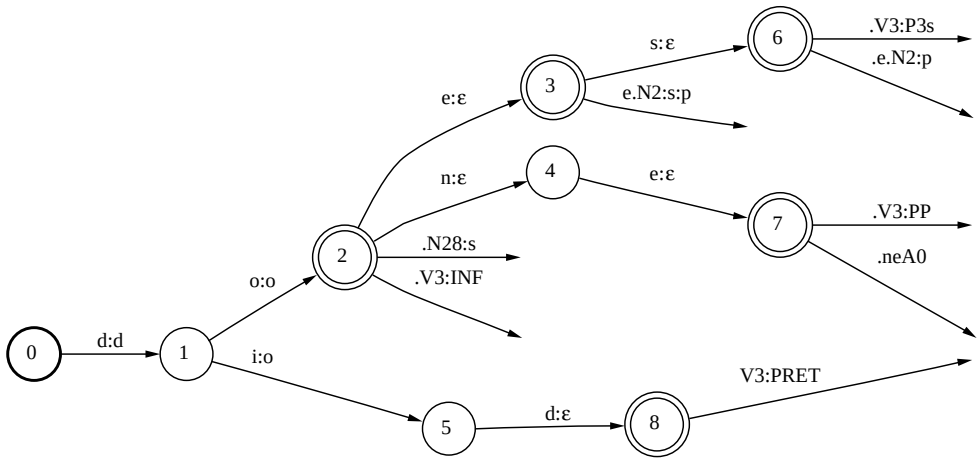


FIG. 14.2: Le dictionnaire sous la forme d'un FST. Figure reprise de (Mohri 1996). Les symboles de sortie sont les analyses morphologiques. Le transducteur est p-sous-séquentiel

Inconvénients. Quel que soit le mode de représentation, la faiblesse du dictionnaire est qu'il ne contient pas l'ensemble des mots du lexique : de nombreux néologismes, la plupart des noms propres et certains mots peu fréquents ne sont pas recensés dans les dictionnaires utilisés.

Un dictionnaire de plusieurs centaines de milliers de mots prend à peine un peu plus d'1 Mo (Mohri 1996) lorsqu'il est représenté sous la forme d'une machine à états finis. Le FSM ne présente donc pas de véritable inconvénient, à moins que l'espace mémoire disponible ne soit fort limité.

La table de hashage peut être encore plus compacte que le FSM, étant donné que la structure de représentation ne nécessite pas la description d'états ni de transitions. Elle peut en outre être plus rapide que le FSM ($O(1)$), parce qu'elle utilise une *clef de hashage*, fonction qui permet de calculer très rapidement la place où une donnée devrait être mémorisée dans la table. La table est par contre de taille fixe, et l'optimisation du temps de recherche

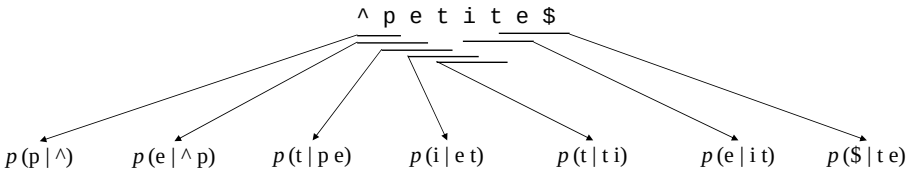


FIG. 14.3: Modèle n-gramme. Le symbole ^ représente le début du mot et $ en représente la fin

dépend fortement de l'adéquation entre la taille de la table et la clef de hashage utilisée. Si la clef de hashage est inadaptée aux données et/ou à la taille de la table, de nombreuses collisions peuvent survenir, et l'algorithme redevient $O(n)$. Enfin, les opérations rationnelles applicables aux FSMs ne s'appliquent pas aux tables de hashage, dont l'utilisation se limite à de simples recherches. Pour de plus amples informations concernant les tables de hashage, le lecteur intéressé se référera utilement à (Aho *et al.* 1974).

14.3.2 Les modèles n-gramme

La notion de *n-gramme* a été expliquée en Section 13.5.2.2. Dans un système de détection des erreurs (Sitar 1961, Harmon 1972), les n-grammes sont faits de caractères (cf. Figure 14.3) et sont évalués soit sur un dictionnaire, soit sur un corpus de textes. En cours d'apprentissage, tout n-gramme $p(c_{i-n+1} \dots c_{i-1}c_i)$ rencontré est mémorisé dans une table avec sa probabilité $p(c_i|c_{i-1} \dots c_{i-n+1})$.

En cours de détection, et pour un mot C donné, les n-grammes mémorisés sont utilisés, mais $P(C)$ n'est jamais calculé, parce que l'unique objectif est de détecter les erreurs dans le mot. Une erreur est détectée si au moins un n-gramme est absent de la table, ou si sa probabilité est en-deçà d'un certain seuil.

Les n-grammes peuvent être *positionnels* ou *non-positionnels*. Le n-gramme positionnel a cet avantage de mémoriser les positions, en termes de lettres, où apparaît un n-gramme. Par exemple, si le corpus d'apprentissage contient les mots *artisan* et *tisser*, le tri-gramme *tis* sera recensé pour les positions 5 (*artisan*) et 3 (*tisser*). Les différentes études réalisées montrent que les n-grammes positionnels donnent des résultats nettement meilleurs (Hanson *et al.* 1976, Hull 1987).

Avantage. Si les modèles n-gramme requièrent généralement soit un dictionnaire, soit un gros corpus de textes afin de pré-calculer la table de n-grammes, ils ont l'avantage de fortement limiter la place mémoire nécessaire lors du traitement (quelques Ko), parce que tout n-gramme qui apparaît plus d'une fois dans le dictionnaire ou le corpus n'est représenté qu'une seule fois, accompagné de sa fréquence et le cas échéant de sa position.

Inconvénient. Qu'il soit positionnel ou non, un modèle n-gramme ne peut jamais assurer que le mot appartient à la langue. Il peut au plus prendre une décision en fonction d'un seuil choisi lors de la modélisation du système. Il arrive donc que des mots existants, mais constitués de n-grammes peu fréquents, soient considérés comme des OOVs, alors que certains OOVs, constitués de n-grammes très fréquents, soient acceptés. Le dictionnaire est sur ce point beaucoup plus fiable, pour autant qu'il soit relativement complet.

14.3.3 Le problème des frontières de mots

Quelle que soit la technique utilisée, la gestion des erreurs qui ont lieu aux frontières des mots reste un problème majeur. Cela est dû à la définition-même de la frontière de mots, classiquement définie comme l'espace. Or, il arrive que plusieurs mots soient accolés (par exemple, « *lemonsieur* ») ou que l'espace séparateur soit mal positionné (par exemple, « *lem onsieur* »).

Il a été montré que près de 15% des erreurs résultant en des OOVs sont dues à la transgression des frontières du mot (Kukich 1992a). Quelques systèmes ont cependant tâché de tenir compte de ce type d'erreurs (Lee *et al.* 1990, Kernighan 1991, Carter 1992, Jones *et al.* 1991). Leur avantage était de travailler sur des vocabulaires limités. Dans ce contexte, deux tests étaient effectués :

1. En ce qui concerne les mots accolés (par exemple, « *lemonsieur* ») la méthode était de tester la possibilité d'insérer un espace entre chaque lettre de la chaîne.
2. Pour ce qui est des mots où l'espace séparateur est mal positionné (par exemple, « *lem onsieur* »), la méthode consistait à estimer un seuil de similarité entre les mots erronés et quelques candidats du lexique.

Dans certains systèmes, ces tests résultent en un treillis de possibilités, dont les différentes solutions sont testées *une à une*.

Quoi qu'il en soit, la détection d'erreurs aux frontières des mots reste l'un des problèmes non résolus en correction orthographique.

14.3.4 Analyse

En général, bien que les n-grammes donnent de bons résultats sur les erreurs générées par la machine (en reconnaissance des caractères, par exemple), l'utilisation de dictionnaires reste la solution la plus efficace pour la détection des erreurs humaines. Les dictionnaires, cependant, sont relativement gourmands en termes de place mémoire.

14.4 Correction des OOVs

Pour la plupart des applications, la simple détection ne suffit pas. En synthèse par exemple, les erreurs présentes dans le texte doivent être détectées, mais aussi corrigées *avant* la génération de la parole.

Pour résoudre ce problème, un grand nombre de méthodes de correction des mots isolés ont été développées. Ces méthodes, généralement, sont contraintes par les caractéristiques des applications pour lesquelles elles ont été mises au point. Elles proposent cependant toutes un processus de correction en trois phases distinctes : détection des erreurs, génération des candidats, classement des candidats.

Les méthodes mises au point peuvent être classées dans six catégories :

1. Le calcul d'une distance d'édition.
2. L'utilisation de clefs de similarité.
3. Les systèmes par règles.
4. Les n-grammes existentiels.
5. Les n-grammes statistiques.
6. Les réseaux de neurones.

Notons que cette organisation est certainement arbitraire, étant donné que certaines de ces catégories se recouvrent partiellement ou partagent des méthodes et des concepts communs.

Dans les pages qui suivent, nous donnons un bref aperçu des principes qui sous-tendent ces différentes techniques. Le lecteur intéressé se reportera utilement à (Peterson 1980, Mitton 1987, Kukich 1992b) pour de plus amples informations. Nous présentons ensuite un condensé des résultats obtenus.

14.4.1 Distance d'édition

La distance d'édition entre deux strings X et Y mesure le nombre minimum d'opérations d'édition nécessaires pour convertir X en Y. Les opérations d'édition sont classiquement la substitution, l'insertion et la suppression d'un symbole, auxquelles certains systèmes ajoutent la transposition de deux symboles adjacents. Cette distance est nommée *distance de Damerau-Levenshtein* du nom de ses auteurs (Damerau 1964, Levenshtein 1966), mais est plus connue sous le nom de *distance de Levenshtein*.

En somme, la distance d'édition modélise exactement les erreurs d'édition présentées en Section 14.1.2.1.

Programmation dynamique. Wagner a été le premier à proposer de résoudre la distance d'édition au moyen de la programmation dynamique (Wagner 1974). Pour une description de la programmation dynamique, nous renvoyons à la Section 13.5.3.4.

Un algorithme efficace est celui proposé par Du & Chang (1992), que nous présentons en Figure 14.4 et que nous illustrons en Figure 14.5.

Pondération. Comme on le voit dans l'algorithme, toute opération d'édition vaut 1. Le résultat de la distance d'édition est donc un entier, équivalent au nombre d'opérations d'édition effectuées.

$$
\begin{array}{rll}
ed(X_i, Y_j) & = ed(X_{i-1}, Y_{j-1}) & \text{si } X_i = Y_j \\
 & & \text{(derniers caractères identiques)} \\
 & = 1 + min\{ed(X_{i-2}, Y_{j-2}, & \text{si } X_{i-1} = Y_j \text{ et } X_i = Y_{j-1} \\
 & \quad X_i, Y_{j-1}, & \text{(transposition des 2 derniers} \\
 & \quad X_{i-1}, Y_j)\} & \text{caractères)} \\
 & = 1 + min\{ed(X_{i-1}, Y_{j-1}, & \text{sinon} \\
 & \quad X_i, Y_{j-1}, & \\
 & \quad X_{i-1}, Y_j)\} & \\
ed(X_0, Y_j) & = j & 0 \leq j \leq n \\
ed(X_i, Y_0) & = i & 0 \leq i \leq m \\
ed(X_{-1}, Y_j) & = \max(m, n) & \text{définition des frontières}
\end{array}
$$

FIG. 14.4: Distance de Levenshtein : algorithme

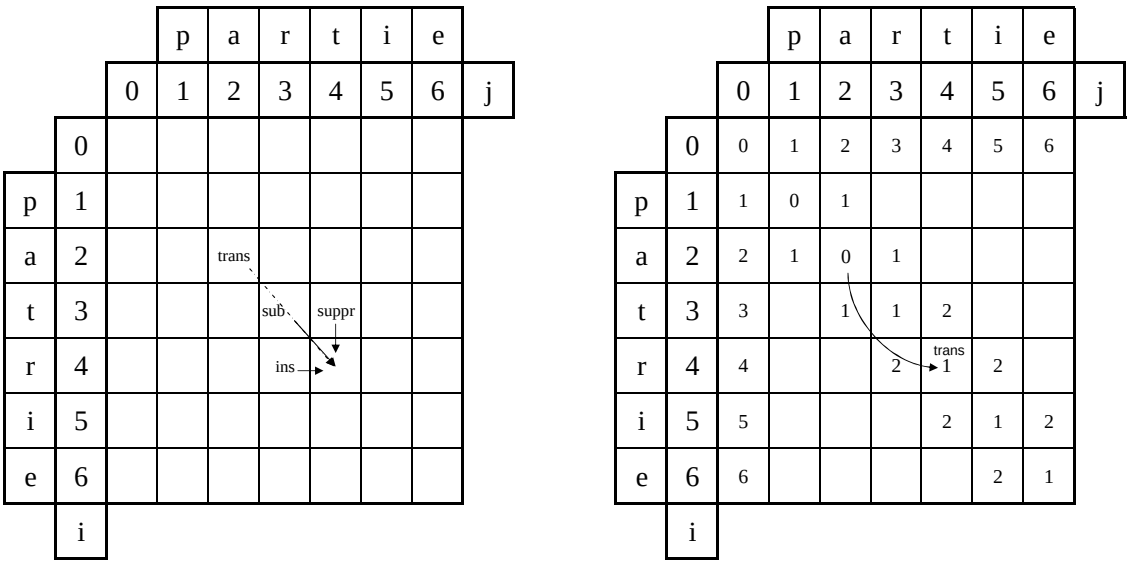


FIG. 14.5: Distance de Levenshtein : illustration

Les systèmes qui adoptent cet algorithme acceptent typiquement une distance d'édition de 1 ou de 2 entre l'OOV x et les candidats y du lexique. Ils se basent sur le fait que l'écart entre une forme erronée et la forme correcte correspondante est rarement supérieur à deux caractères (cf. Section 14.1.2).

La liste des candidats étant constituée, la probabilité d'un candidat y est ensuite simplement estimée comme suit :

$$p(y|x) = \begin{cases} \alpha & \text{si } y = x \\ \frac{1-\alpha}{|\{y\}|-1} & \text{sinon} \end{cases} \quad (14.4.1.1)$$

où α est un paramètre défini empiriquement (typiquement, $\alpha = 0,99$), $\{y\}$ est l'ensemble qui comprend l'OOV x et ses corrections, et $|\{y\}|$ représente le nombre d'éléments de l'ensemble.

Inconvénients. Implémentée sous cette forme, la distance de Levenshtein comporte trois défauts majeurs :

1. La pondération proposée rend tous les candidats équiprobables. En outre, plus le nombre de candidats est grand ($|\{y\}|$), plus le poids d'un candidat donné diminue. La probabilité de choisir une correction plutôt que la forme erronée diminue donc également.
2. Le nombre d'erreurs ne dépend en aucun cas de la longueur du mot, alors qu'il a été établi que la longueur du mot influe sur le nombre d'erreurs introduites (cf. Section 14.1.2).
3. La distance s'établit *au vol* entre le mot supposé erroné et chaque mot du dictionnaire. Or, de nombreux mots partagent un préfixe commun. Le nombre des calculs pourrait donc être considérablement réduit.

Pour remédier aux inconvénients de la distance de Levenshtein classique, quelques méthodes d'optimisation ont été proposées.

14.4.1.1 Distance pondérée

Dans ce modèle (Church & Gale 1991), chaque erreur d'édition est pondérée individuellement en fonction du contexte. L'entraînement de ces pondérations est réalisé à l'aide d'un corpus dont les auteurs ont prélevé l'ensemble des mots n'apparaissant pas dans leur dictionnaire. De cet ensemble ne sont conservés que les mots qui ne sont pas éloignés de plus d'une erreur d'édition d'un mot valide. Les probabilités des erreurs d'édition sont ensuite apprises à partir de ce corpus aligné {erreurs, corrections}. A partir de cet apprentissage, le

modèle détermine la probabilité d'un OOV x étant donné un candidat y :

$$p(x|y) = \begin{cases} del(y_{i-1}, y_i)/c(y_{i-1}, y_i) & \text{supression} \\ ins(y_{i-1}, x_i)/c(y_{i-1}) & \text{insertion} \\ sub(x_i, y_i)/c(y_i) & \text{substitution} \\ trans(y_i, y_{i+1})/c(y_i, y_{i+1}) & \text{transposition} \end{cases} \tag{14.4.1.2}$$

où :

- $c(a, b)$ et $c(a)$ représentent respectivement le nombre d'occurrence de ab et de a dans le corpus.
- $del(a, b)$ représente le nombre de fois où ab ont été tapés a dans le corpus.
- $ins(a, b)$ représente le nombre de fois où a a été tapé ab dans le corpus.
- $sub(a, b)$ représente le nombre de fois où b a été tapé a dans le corpus.
- $trans(a, b)$ représente le nombre de fois où ab ont été tapés ba dans le corpus.

Par rapport à la distance classique, ce modèle a pour seul avantage de distinguer les candidats à l'aide de poids différents, qui tiennent compte de la fréquence de l'erreur qui les séparent de l'OOV. Pour des raisons d'optimisation, l'approche accepte cependant une seule erreur d'édition entre l'OOV et ses candidats, quelle que soit la longueur de l'OOV.

14.4.1.2 Modèle de langue

L'idée est d'outrepasser les quatre opérations de base définies sur les caractères et de proposer un modèle de langue qui estime la probabilité, pour $x, y \in \Sigma^*$, de taper x à la place de y (Brill & Moore 2000). Le modèle, selon la métaphore du canal bruité et en appliquant le théorème de Bayes, se formule :

$$P(y|x) = P(x|y) \cdot P(y) \tag{14.4.1.3}$$

où $P(y)$ modélise la source tandis que $P(x|y)$ modélise le canal.

Nous nous concentrons ici sur la modélisation du canal : $P(x|y)$ implique de déterminer la meilleure manière d'aligner l'OOV x et un candidat y. Il faut donc déterminer leur meilleure segmentation :

$$P(x|y) = \max_{X \in Seg(x), Y \in Seg(y)} \prod_{i=1}^{|Y|} p(X_i|Y_i) \tag{14.4.1.4}$$

où $Seg(x)$ (resp. y) est l'ensemble de toutes les segmentations possibles de x (resp. y). Ce modèle peut également tenir compte de la position de la string dans le mot, auquel cas la probabilité d'association de deux sous-strings se détermine :

$$p(X_i|Y_i, \lambda) \tag{14.4.1.5}$$

où λ est le début, le milieu ou la fin du mot. Par exemple, on peut supposer que la confusion *ent*/*ant* sera plus fréquente en fin de mot (*patiemment*/*patiemmant*).

La meilleure segmentation est initialement déterminée à partir du corpus d'entraînement. Le principe est de calculer la distance d'édition classique entre l'OOV et sa correction, et d'intégrer les substitutions et les insertions aux caractères précédents ou suivants, comme dans l'exemple suivant :

a	c		t	u	a	l
a	k	g	s	u	a	l

où l'on peut proposer plusieurs regroupements, comme par exemple $c \rightarrow kg$ ou $ct \rightarrow kgs$. Ces diverses solutions sont pondérées sur l'ensemble du corpus.

Ce modèle a très clairement l'avantage d'autoriser des opérations d'édition de n caractères vers m caractères, contrairement aux modèles précédents qui n'autorisaient que des opérations 1-1. Les auteurs ont réalisé des tests où n et m varient indépendamment de 1 à 5 caractères.

Le seul inconvénient de ce modèle est sa forte dépendance au contexte : un caractère qui serait confondu dans un contexte différent de ceux observés sur le corpus ne serait pas corrigé.

14.4.1.3 Modèle de langue augmenté de phonétique

Nous avons précédemment expliqué que certains OOVs sont le résultat d'erreurs phonétiques ou cognitives. Or, le modèle précédent ne gère pas correctement ces erreurs. Par exemple, il corrige l'OOV *edelvise* (*edelweiss*) par *advise*. C'est ce qui motive cette nouvelle approche (Toutanova & Moore 2002), qui se base sur la précédente et la complète d'un modèle phonétique.

L'idée est simple. En plus de l'estimation sur corpus des probabilités de substitution typographique $P(x|y)$, cette approche étudie également les probabilités de substitution phonétique $P_{ph}(x|y)$. Les deux probabilités sont combinées dans la prise de décision :

$$\log P_{CMB}(x|y) = \log P(x|y) + \lambda \log P_{ph}(x|y) \tag{14.4.1.6}$$

où la probabilité de substitution phonétique est évaluée comme suit :

$$P_{ph}(x|y) = \sum_{pho(y)} \frac{1}{|pho(y)|} \max_{pho(x)} \left(P(pho(x)|pho(y)) \cdot P(pho(x)|x)\right) \tag{14.4.1.7}$$

où $pho(x)$ retourne la phonétisation de la string x.

La probabilité de substitution $P(pho(x)|pho(y))$ d'une string phonétique par une autre est évaluée comme dans le modèle de Brill & Moore (2000), sur le corpus d'entraînement. Ceci demande donc de phonétiser à la fois les IVs (y) et les OOVs (x). Les IVs sont retranscrits à l'aide d'un dictionnaire phonétique.

Pour les OOVs, les auteurs ont employé un modèle n-gramme entraîné sur un corpus phonétisé. Le principe est d'estimer l'ensemble des phonétisations possibles pour une lettre cible L_c dans le contexte du graphème gauche Gr_g et du graphème droit Gr_d :

$$L_c \rightarrow \{pho_1/w_1, pho_2/w_2 \ldots pho_n/w_n\} \; :: Gr_g \_ Gr_d \tag{14.4.1.8}$$

où pho_i est la i^e phonétisation de L_c à laquelle l'apprentissage a accordé le poids w_i. Notons que c'est la multiplication de ces poids pour un mot donné qui détermine $P(pho(x)|x)$.

Le modèle rend ainsi possible les substitutions n-m au niveau typographique et au niveau phonétique. Ce modèle est certainement la distance d'édition la plus aboutie.

Le seul inconvénient de ce modèle est que les conversions phonétiques sont calculées en cours de traitement.

14.4.1.4 Distance d'édition modélisée par machines à états finis

L'un des inconvénients majeurs de la distance de Levenshtein dans son implémentation classique est qu'elle doit être complètement recalculée pour chaque mot du dictionnaire. Ce caractère éminemment séquentiel rend cette distance inutilisable en pratique, dès que la taille du dictionnaire atteint quelques milliers de formes. Or, un dictionnaire un tant soit peu complet compte plusieurs centaines de milliers de formes.

Pour remédier à ce problème, certains chercheurs ont proposé de modéliser le comportement de l'algorithme au moyen de machines à états finis.

Méthodes proposées. Les méthodes les plus efficaces sont celle d'Oflazer (1996), qui propose un parcours d'automate tolérant aux erreurs, et celle de Schulz & Mihov (2002), qui définissent un nouveau type d'automate, l'automate de Levenshtein.

Automate tolérant aux erreurs. Chez Oflazer (1996), la distance de Levenshtein est directement calculée pendant le parcours d'une string dans l'automate. L'algorithme est présenté en Pseudocode 33.

Le principe de l'algorithme est de parcourir l'automate dans une recherche en profondeur d'abord, guidé par la string u à reconnaître. L'algorithme utilise une pile P de paires (v, p), où v est une string construite à partir des symboles des transitions parcourues, et p est l'état atteint. Une paire (v, p) n'est ajoutée dans la pile que si un test l'autorise : le *cutoff*, qui compare les strings u et v. La taille de la pile et l'ampleur de la recherche en profondeur sont donc contrôlées par ce test. La string v n'est en outre considérée comme une correction potentielle que si v est une mot du langage (l'état de l'automate que v atteint est final) et si la distance d'édition $ed(u, v)$ est inférieure ou égale au seuil fixé max.

Le *cutoff* [3], réalisé entre la string à reconnaître u et la string candidate v, détermine la *distance d'édition minimale* entre v et certaines sous-strings de u. Les sous-strings de u

[3] *Cutoff* : limite, seuil.

concernées par le cutoff sont celles dont la longueur se situe entre $l_1 = \max(1, |v| - max)$ et $l_2 = \min(|u|, |v| + max)$. Par exemple, pour un seuil d'édition valant 2, si la string à reconnaître est *élèbve*, et que le candidat en cours de construction est *élèv*,

- $l_1 = \max(1, 4 - 2) = 2$
- $l_2 = \min(6, 4 + 2) = 6$

A partir de l_1 et l_2, le *cutoff* se calcule comme suit :

$$\begin{aligned} cuttoff(\text{élèbve,élève}) = \min\{&ed(\text{él,élèv}) = 2 \qquad = 1 \\ &ed(\text{élè,élèv}) = 1 \\ &ed(\text{élèb,élèv}) = 1 \\ &ed(\text{élèbv,élèv}) = 1 \\ &ed(\text{élèbve,élèv}) = 2\} \end{aligned}$$

Le seuil d'édition étant fixé à 2 et le cutoff valant 1, la string candidate *élèv* peut être ajoutée à la pile. Par contre, cette string n'est pas un mot du langage : elle n'est donc pas ajoutée à l'ensemble des corrections.

La méthode, efficace, présente cependant deux inconvénients. Premièrement, le *cutoff* est coûteux, parce qu'il doit être réalisé à chaque étape de l'avancement dans la construction d'un candidat. Deuxièmement, le calcul de la distance d'édition reste nécessaire, et est réalisé chaque fois qu'un état final est atteint.

Automate de Levenshtein. L'objectif de cette approche est d'éviter le calcul du *cutoff* et de la distance d'édition, afin d'améliorer le temps de parcours de l'automate représentant le dictionnaire. L'idée est ici de générer un automate qui représente l'ensemble des strings v, construites à partir de la string u à reconnaître, qui ne dépasse pas un seuil d'édition max. Cet automate est qualifié d'*automate de Levenshtein*.

Lorsque cet automate est construit, le processus de recherche des corrections devient trivial : il se limite au calcul de l'intersection entre l'automate de Levenshtein et l'automate représentant le dictionnaire.

Le cœur de cet algorithme est donc la construction de l'automate de Levenshtein, pour une string u à reconnaître. Les auteurs démontrent qu'il est possible de précalculer un *patron d'automate*, à partir duquel l'automate de Levenshtein pour une string u donnée peut être généré automatiquement. La longueur de la démonstration dépasse largement le cadre de cet état de l'art, raison pour laquelle nous renvoyons le lecteur à l'article cité. La démonstration prouve que le patron d'automate dépend uniquement de l'alphabet du langage et de la valeur du seuil d'édition. Un patron différent doit donc être pré-calculé pour chaque seuil d'édition désiré.

Les auteurs signalent que, malheureusement, la taille du patron augmente exponentiellement par rapport à la taille du seuil. Cette méthode n'est donc applicable que pour de petits seuils d'édition. Il faut cependant reconnaître qu'en pratique, de grands seuils

Require: $A = (\Sigma, Q, i, F, E)$, un dictionnaire représenté sous la forme d'un automate,
u, la string à reconnaître,
max, la distance d'édition maximale.
Ensure: V, l'ensemble des strings v de A pour lesquelles $ed(u, v) \leq max$

$V \leftarrow \emptyset$
PUSH$(P, (\epsilon, i))$
while $P \neq \emptyset$ **do**
  $(v, p) \leftarrow$ POP(P)
  for each $(p, a, q) \in E$ **do**
    $v' \leftarrow v \cdot a$
    PUSH$(P, (v', q))$ **if** $cutoff(u, v') \leq max$
    ADD(V, v') **if** $q \in F$ **and** $ed(u, v') \leq max$
  end for
end while
return V

Pseudocode 33: Distance d'édition d'Oflazer (1996)

Système	1 erreur	2 erreurs	3 erreurs
Oflazer (1996)	15	95	349
Schulz & Mihov (2002)	2,2	30	160

TAB. 14.3: Temps de traitement (en ms) pour trouver les candidats dans le dictionnaire

d'édition sont rarement utilisés, étant donné le nombre de corrections fort éloignées qu'ils permettent de retrouver.

Analyse. Ces deux méthodes atteignent leur objectif, qui est de limiter le temps de traitement nécessaire (cf. Table 14.3). Elles sont donc efficaces.

Elles présentent un seul véritable inconvénient : elles emploient des machines à états finis, mais ne peuvent s'inscrire dans un processus exclusivement régi par des machines à états finis. Ceci est dû aux deux caractéristiques suivantes :

1. Le seuil d'édition ne dépendra de la longueur de la string à reconnaître qu'au prix d'un calcul préliminaire, réalisé *hors de l'automate*.

2. La forme à reconnaître n'est pas représentée sous la forme d'un automate, mais d'une string. Dans le cas d'Oflazer (1996), les candidats sont en outre également représentés sous la forme de strings.

14.4.2 Clefs de similarité

L'idée est de faire correspondre une clef à chaque chaîne de caractères de sorte que les chaînes contenant les mêmes caractères aient une clef identique (Davidson 1962, Pollock & Zamora 1984). Ainsi, la clef calculée pour une chaîne mal orthographiée fournira un pointeur vers toutes les chaînes similaires dans le lexique : les candidats.

De manière générale, la technique consiste à répartir l'alphabet en n sous-ensembles, chaque sous-ensemble se voyant attribuer un entier. La Table 14.4 présente la répartition du premier système du genre, SOUNDEX, dont la première version date de 1918.

a, e, i, o, u, h, w, k, y	→	0
b, f, p, v	→	1
d, t	→	2
c, g, j, k, q, s, x, z	→	3
l	→	4
m,n	→	5
r	→	6

TAB. 14.4: Clefs de similarité

L'exemple de la table s'applique à l'anglais américain. Seule la première lettre du mot est conservée. Toutes les autres sont remplacées par leur code. Comme on le constate dans la table présentée, les voyelles sont supprimées, ainsi que w et h. Les consonnes sont regroupées, en fonction de leur prononciation courante, en ensembles que l'on peut considérer comme des classes phonétiques.

Lorsqu'une clef a été attribuée à chaque mot du dictionnaire, celui-ci est réordonné sur les clefs. La recherche des mots candidats pour une forme erronée se déroule alors comme suit : une clef est calculée pour le mot erroné, et localisée dans le lexique de clefs triées. Les candidats sont alors les termes correspondant à la clef elle-même, ainsi que ceux dont les clefs appartiennent à un intervalle donné autour de la clef recherchée.

De nombreuses évolutions ont été proposées au cours des années et en fonction de la langue, mais le principe a été conservé.

Analyse. Ce système a montré son intérêt, et ce principalement dans l'étude de l'évolution des patronymes. Il présente cependant trois inconvénients dans le cadre de la correction orthographique :

1. La première lettre est conservée, ce qui implique que les erreurs sur la première lettre du mot ne sont pas gérées. Or, bien qu'elles ne soient pas les plus fréquentes, les erreurs sur la première lettre sont néanmoins une réalité. C'est dans ce contexte qu'interviennent les problèmes d'espaces aux frontières des mots.
2. Les voyelles sont systématiquement confondues. Or, ces lettres ne sont pas particulièrement proches sur le clavier et se prononcent souvent de manière fort différente. Il est

donc malheureux de considérer comme proches des termes comme *malle* et *mille*, que tout sépare : la distance sur le clavier, la phonétique et la sémantique.

3. La pondération des candidats est délicate : tous les termes référencés par une même clef sont considérés comme équivalents, et l'estimation de la distance entre deux clefs n'est pas spécialement représentative de la distance entre les mots référencés par ces clefs.

14.4.3 Systèmes par règles

Il s'agit d'algorithmes ou d'heuristiques qui modélisent la correction orthographique sous la forme de règles de conversion *forme erronée* → *forme correcte*. Les exemples que nous donnons, en anglais, sont repris des systèmes présentés (Yannakoudakis & Fawthrop 1983, Means 1988).

L'approche classique consiste à commencer par une analyse morphologique des suffixes éventuels appartenant à un OOV (par exemple, *ed* dans *pluged*, *ing* dans *seperating*). Si la recherche donne un résultat, le suffixe est supprimé de l'OOV et une recherche en terme de distance d'édition est réalisée dans un dictionnaire de lemmes à partir du terme tronqué (*plug-*, *seperat-*). La distance autorisée vaut 1. Si la recherche donne un résultat, le système s'interrompt. Notons que les systèmes présentés signalent qu'il est difficile de traiter ce que nous appellerons les « lemmes tronqués » (*seperat-*), parce qu'ils s'éloignent d'une distance supérieure à 1 des lemmes corrects (*sep<u>a</u>rat<u>e</u>*) [4].

Lorsque cette analyse morphologique échoue, une seconde approche est tentée : la détection d'abréviations. Des règles de bonne formation déterminent si l'OOV est un mot tronqué (*spr* -> *spring*, *spray*, etc.) ou correspond à un squelette consonantique (*spr* peut venir de *spare*, *super* ou *speaker*, mais pas de *separate* ni de *sport*). Les règles sont réparties en catégories et déterminent un degré de plausibilité linguistique, sur la base duquel la pondération des solutions est proposée.

Si cette seconde analyse échoue également, une distance d'édition est alors calculée. Cependant, cette distance d'édition ne correspond pas à une distance de Levenshtein classique. Ici, les distances acceptées représentent toutes des phénomènes linguistiques, comme la réduction ou la création de diphtongues vocaliques et de géminées consonantiques.

En somme, le classement des candidats est obtenu en fonction d'un poids accordé à chaque règle, appliquée seulement si la règle précédente n'a donné aucune solution. Le poids d'une règle correspond à une estimation linguistique de la probabilité que l'erreur décrite se produise.

[4] Bien que cela n'ait pas été proposé dans la littérature, le système pourrait ici faire l'hypothèse du suffixe à ajouter à la racine pour trouver le lemme hypothétique : *seperating* - *ing* + *e* = *seperate*. C'est ce que nous faisions dans la toute première version de notre analyseur morphologique (cf. Chapitre 13). Cette procédure ne s'appliquait alors qu'aux mots corrects, parce que l'analyseur n'incluait pas de correction orthographique. La procédure serait cependant applicable au cas des OOVs.

Inconvénients. Les systèmes par règles sont finement décrits et requièrent des experts capables de modéliser la langue à traiter. Les règles sont dépendantes de la langue, et doivent être redéfinies pour toute nouvelle langue à traiter.

Ces systèmes s'appliquent en outre plus facilement à des domaines limités, pour lesquels les erreurs potentielles peuvent être relativement facilement modélisées, étant donné que le lexique et la syntaxe sont moins vastes, et que les types d'erreurs possibles sont moins nombreux.

14.4.4 *n*-grammes existentiels

Comme nous l'avons expliqué en Section 14.3.2, l'approche *n*-gramme se base sur une phase d'apprentissage, au cours de laquelle des *n*-grammes de lettres, typiquement des tri-grammes, sont calculés sur un dictionnaire. La méthode diffère cependant ici dans le sens où l'objectif est uniquement de déterminer la liste des *n*-grammes existants : l'occurrence des *n*-grammes n'est pas prise en compte par le modèle (Riseman & Hanson 1974, Ullmann 1977, Angell *et al.* 1983).

Une erreur dans un mot est détectée si au moins un *n*-gramme est inconnu. Cependant, pour localiser l'erreur de manière précise, il est nécessaire que plusieurs *n*-grammes successifs soient inconnus : l'erreur est alors posée à leur intersection.

La correction est proposée sans recours au dictionnaire : la technique consiste à tester les *n*-grammes existants et proches des *n*-grammes erronés, jusqu'à ce qu'une combinaison valide soit trouvée. La méthode est fort limitée, puisqu'elle ne gère que les substitutions.

Avantage. Cette technique offre un avantage de taille : elle permet d'éviter une recherche exhaustive sur l'ensemble du dictionnaire.

Inconvénients. Un inconvénient mineur est qu'il est possible qu'une correction résulte en un OOV, mais ceci à de rares occasions. Un inconvénient plus gênant est que la technique n'est prévue que pour les erreurs de substitution et ne semble pas facilement adaptable aux autres types d'erreurs (Kukich 1992b).

14.4.5 *n*-grammes statistiques

Contrairement à l'approche précédente, celle-ci tient compte de l'occurrence des *n*-grammes dans le corpus d'apprentissage (Hanson *et al.* 1976, Kahan *et al.* 1987, Jones *et al.* 1991, Coker *et al.* 1990). Deux types de probabilités ont été exploitées : les probabilités de transition et celles de confusion.

1. Une erreur est détectée en fonction des probabilités de transition. Nous avons décrit ce principe en Section 14.3.2.
2. Les probabilités de confusion sont utilisées en phase de correction. Elles sont dépendantes de la *source* : elles estiment l'occurrence de la confusion entre deux lettres.

Bien sûr, la probabilité de confusion dépend de l'outil : un système de reconnaissance des caractères ne confondra pas les mêmes lettres qu'un utilisateur entrant des données au clavier. Dans les deux cas, une phase d'apprentissage permettra d'estimer ces probabilités de confusion :

- Dans le cas du système de reconnaissance des caractères, on établira ces probabilités par comparaison de l'étiquette associée à un caractère dans la base d'apprentissage et de l'étiquette attribuée au caractère par le système en phase de reconnaissance.
- Les erreurs d'édition seront estimées à partir de corpus de textes entrés au clavier et présentant des fautes d'orthographe.

Les recherches en reconnaissance de caractères ont montré que ces probabilités seules ne permettent pas d'atteindre un taux de correction suffisant. Par contre, la combinaison de ces probabilités et de dictionnaires donne de très bons résultats. L'algorithme général est le suivant :

1. Une première phase attribue, à chaque lettre de l'OOV Y en cours d'analyse, un vecteur de 26 probabilités, une par lettre de l'alphabet.
2. Une seconde phase utilise un dictionnaire, de manière à restreindre le choix du mot correct X à un mot du dictionnaire.
3. La troisième phase cherche à déterminer le mot X qui correspond effectivement à l'OOV Y, ce qui est modélisé à l'aide de la métaphore du canal bruité (cf. Section 4.1) :

$$X = \arg\max_X P(X|Y) \tag{14.4.5.1}$$

ce qui, par le théorème de Bayes (Boîte *et al.* 2000), se résout classiquement par :

$$P(X|Y) = \arg\max \frac{P(Y|X)P(X)}{P(Y)} \tag{14.4.5.2}$$

où $P(X|Y)$ est la probabilité conditionnelle que X soit le mot correct, $P(Y|X)$ est la probabilité conditionnelle d'observer Y quand X est le mot correct, et $P(X)$ ainsi que $P(Y)$ sont les probabilités indépendantes des mots X et Y. $P(Y)$ étant une constante, quel que soit le mot X en cours d'étude, l'équation peut être réduite à :

$$P(X|Y) = \arg\max P(Y|X)P(X) \tag{14.4.5.3}$$

où $P(X)$ est le maximum de vraisemblance du mot X estimé sur un corpus d'apprentissage, et $P(Y|X)$ est le produit des probabilités de confusion des caractères individuels de X et Y :

$$P(Y|X) = \prod_{i=1}^{m} P(Y_i|X_i) \tag{14.4.5.4}$$

Inconvénient. Les besoins computationnels de cet algorithme croissent linéairement par rapport à la taille du dictionnaire. De ce fait, même en partitionnant celui-ci en fonction de la longueur des mots, le traitement de gros dictionnaires est impossible en pratique.

Ceci a conduit certains chercheurs à se passer du recours au dictionnaire. Le coût de cette simplification est évidemment une baisse considérable des résultats obtenus.

14.4.6 Réseaux de neurones

14.4.6.1 Principe

Les réseaux de neurones trouvent leur origine dans les récentes études du système nerveux central humain, dont ils constituent une pâle imitation : ils modélisent l'activité des neurones cérébraux et les connexions synaptiques qui les relient.

Un réseau de neurones est constitué de couches, chacune étant faite d'un certain nombre de nœuds. Chaque nœud d'une couche donnée possède une *activation* [5] et est relié à l'ensemble des nœuds de la couche suivante par une connexion pondérée [6].

De ce réseau multicouche, seules les première et dernière couches sont visibles. L'une constitue l'entrée du réseau et l'autre, la sortie. Les autres couches sont dites *cachées*. Un réseau typique compte trois couches, dont une seule couche cachée. Les valeurs des nœuds des couches d'entrée et de sortie sont

- Soit des valeurs binaires : 1 pour actif et 0 pour inactif.
- Soit des valeurs continues : elles indiquent dans ce cas qu'un nœud est plus ou moins actif. Notons que, dans ce cas, un nœud est généralement favorisé, et reçoit plus de 90% du taux de confiance. De ce fait, l'activation des autres nœuds devient rapidement non significative.

Un réseau de neurones est généralement entraîné par rétro-propagation (Rumelhart *et al.* 1986), dont le principe est le suivant. Les poids du réseau sont initialisés à des valeurs faibles choisies de manière aléatoire. Ensuite, chaque paire {*entrée erronée*, *sortie correcte*} du corpus d'entraînement est présentée au réseau. L'entrée transite au travers du réseau qui, soit active en sortie le nœud de la forme supposée correcte, soit propose en sortie une pondération pour chaque nœud de la couche. La sortie obtenue est comparée à la sortie souhaitée : une différence est calculée pour chaque nœud de la couche de sortie. Cette différence permet de réajuster le poids de chaque nœud par rétro-propagation. Ce processus est répété de manière itérative jusqu'à ce que tous les poids du réseau convergent. Le résultat est un ensemble de poids qui produisent une pondération de sortie proche de la sortie désirée pour chaque entrée de l'ensemble d'entraînement, ainsi que pour des entrées *similaires* à celles de l'ensemble d'entraînement, mais absentes de celui-ci. Cette dernière caractéristique est considérée comme la capacité du réseau à généraliser.

[5] Une valeur réelle entre 0 et 1.
[6] Une valeur réelle entre $-\infty$ et $+\infty$.

14.4.6.2 En correction

Les réseaux de neurones sont des candidats logiques à la correction orthographique, étant donné leur capacité intrinsèque à gérer des entrées incomplètes et bruitées : ils peuvent dès lors être directement entraînés sur des corpus d'erreurs.

Dans les systèmes de correction qui emploient des réseaux de neurones (Kukich 1988, Cherkassky & Vassilas 1989, Deffner *et al.* 1990), les entrées du réseau sont représentées par des vecteurs de n-grammes binaires contenant une ou plusieurs erreurs. La sortie correspondante est un vecteur de m éléments, où m est le nombre de mots contenus dans le lexique. Deux résultats sont possibles, selon le type de réseau utilisé. Soit seul le nœud de sortie correspondant au mot correct est activé, soit tous les nœuds reçoivent un poids, les mots les plus probables étant les mieux pondérés.

14.4.6.3 Analyse

Avantages. Le système est incontestablement très rapide : le parcours d'un réseau est instantané. En outre, moyennant un réentraînement périodique, le réseau peut petit à petit modéliser les erreurs de l'utilisateur.

Inconvénients. La méthode se base sur un apprentissage qui utilise un corpus associatif de paires {n-grammes erronés, mot correct}. Par définition, le système est donc limité à ne corriger que les erreurs qu'il a rencontrées au cours de l'apprentissage, ainsi que des erreurs absentes du corpus, mais similaires à celles du corpus. Or, les erreurs réalisées par les utilisateurs sont par nature aléatoires, ce qui implique que de *nouvelles erreurs* peuvent toujours être rencontrées.

Un autre inconvénient du système est qu'il est difficile de comprendre *comment* ou *pourquoi* une correction a été réalisée. Ce type de système est donc difficilement paramétrable.

Un dernier inconvénient est que le réseau gère difficilement les lexiques fort étendus, étant donné qu'à chaque mot doit correspondre un nœud de la couche de sortie. Le réseau de neurones semble donc plus adapté aux applications à petit vocabulaire.

14.4.7 Evaluation des approches présentées

Evaluation 1. La Table 14.5 présente l'évaluation réalisée par Kukich (1992b), qui concerne toutes les méthodes sauf les améliorations apportées à la distance d'édition, plus récentes. Cette évaluation a été réalisée sur un ensemble de 170 OOVs et sur la base d'un dictionnaire de 1142 mots. Ce test semble représentatif de la répartition « naturelle » des erreurs, étant donné que 25% des OOVs contiennent plusieurs erreurs et que 65% sont des mots courts.

Sur ce test, on constate que la distance d'édition ne procure pas les meilleurs résultats, alors que toutes les autres méthodes semblent se valoir. Ce constat peut être surprenant, mais s'explique cependant. La distance d'édition classique propose au mieux

une modélisation de ce qui se passe au niveau du clavier de l'utilisateur. C'est là tout le contexte qu'elle propose. Elle ne tient compte ni des approximations phonétiques, ni du contexte de l'erreur commise. Les autres méthodes, par contre, utilisent toutes un contexte plus ou moins étendu.

Les autres méthodes présentent cependant les inconvénients suivants :

1. Les clefs de similarité ne traitent pas la première lettre du mot, et confondent toutes les voyelles alors que l'être humain ne confond, lui, que les groupes vocaliques de prononciation proche.
2. Les systèmes par règles appliquent les possibilités de correction de manière séquentielle : le système s'arrête dès qu'une solution est trouvée. Ces systèmes sont en outre fortement dépendants de la langue et de l'expertise de leurs concepteurs.
3. Les modèles n-gramme sont plus performants lorsqu'ils sont combinés à des dictionnaires, mais ces derniers ralentissent considérablement le système, ce qui a poussé les chercheurs à supprimer le recours aux dictionnaires. La conséquence est que les décisions du système paraissent fort aléatoires, et ne sont plus gérées que par les n-grammes du corpus qui a servi à l'entraînement.
4. Les réseaux de neurones se limitent à la modélisation des erreurs rencontrées au cours de l'entraînement, sont difficilement paramétrables et sont plus adaptés aux petits vocabulaires.

Si l'on exclut les réseaux de neurones, les systèmes imposent en outre la même distance maximale entre l'OOV et sa correction, quelle que soit la longueur de l'OOV envisagé. Ceci n'est certainement pas pertinent, puisque, comme nous l'avons vu, le nombre d'erreurs est fortement dépendant de la longueur du mot.

Evaluations 2 et 3. Deux tests permettent d'évaluer la pertinence des améliorations apportées à la distance d'édition classique. La Table 14.6, qui condense les résultats décrits dans (Brill & Moore 2000), compare la distance pondérée et le premier modèle de langue. Dans cette évaluation, le modèle de langue autorise des substitutions allant de 1 à 3 caractères. La Table 14.7, réalisée à partir des résultats présentés dans (Toutanova & Moore 2002), compare le modèle de langue sans et avec phonétique. On constate que les trois méthodes sont pertinentes. La meilleure méthode est cependant la distance qui tient compte des similarités phonétiques, ce qui justifie l'ajout de cette information.

On constate que les résultats sont excessivement bon si l'on considère les trois ou quatre meilleures *propositions* du système. Cependant, ces propositions ne peuvent devenir des *corrections* que si une décision est prise par le système. La correction des OOVs, pour ce faire, doit être intégrée dans un système gérant également les erreurs dépendantes du *contexte*.

Méthode	**Taux (%)**
Distance d'édition	62
Clefs de similarité	**78**
n-grammes existentiels	75
n-grammes statistiques	**78**
Réseaux de neurones	75

TAB. 14.5: Evaluation 1

Méthode	**Fenêtre max.**	**1-best**	**2-best**	**3-best**
Distance d'édition	-	89,5	94,9	96,5
Modèle de langue	1	90,9	95,6	96,8
	2	92,9	97,1	98,1
	3	93,6	97,4	98,5

TAB. 14.6: Evaluation 2

Méthode	**1-best**	**2-best**	**3-best**	**4-best**
Modèle de langue	94,21	98,18	98,90	99,06
Modèle phonétique	95,58	98,90	99,34	99,50

TAB. 14.7: Evaluation 3

14.5 Correction des erreurs sur IVs

La correction des erreurs sur IVs est une nécessité : 25 à 50% des erreurs relevées dans les textes sont de ce type (Kukich 1992b). Nous rappelons qu'une erreur sur IV consiste à utiliser un mot appartenant à la langue dans le mauvais contexte. En voici quelques exemples :

1. Les erreurs syntaxiques (*il dois* → *il doit*, *je serais* ↔ *je serai*).
2. Les erreurs syntaxico-sémantiques, dues à des erreurs d'édition (*tu lui par les* → *parles*).
3. Les erreurs sémantiques, réparties entre erreurs d'édition (*forme* ↔ *firme*) et erreurs lexicales (*sceptique* ↔ *septique*, *palais* ↔ *palet*).
4. Les erreurs pragmatiques (*Leonhard Euler dit en 1961* → *1761*).

Afin de pouvoir à la fois détecter et corriger ces erreurs, il semble donc nécessaire de recourir à des informations provenant d'un contexte plus ou moins large. Ces informations peuvent en outre être également utiles à la correction des OOVs, puisque les méthodes présentées précédemment permettent de créer des listes plus ou moins longues de candidats, mais sont contraintes, sans information contextuelle, à choisir le plus probable *a priori*.

Les approches suivantes ont été proposées dans le domaine :

1. La construction de règles linguistiques à orientation lexicale, syntaxique ou sémantique.
2. Le recours à un modèle de langue à orientation lexicale.
3. La gestion de listes de confusion.
4. L'extraction de caractéristiques multi-niveaux.
5. L'exploitation des ressources du Web.

Une analyse globale de la correction en contexte suit les descriptions des différentes approches.

14.5.1 Construction de règles linguistiques

Note 14.5.1. Nous rappelons que la Section 13.5 a fait le point sur les principes généraux de l'analyse syntaxique automatique basée sur des règles linguistiques. Ce module syntaxique occupe une place très importante dans les systèmes linguistiques, parce qu'il constitue le dernier niveau abouti de la chaîne d'analyse : l'état de l'art en analyse sémantique et pragmatique ne permet actuellement que le traitement de domaines restreints. La présentation des modèles de correction par règles linguistiques s'appuie sur les caractéristiques de l'analyse linguistique relevées en Section 13.5.

Les chercheurs en traitement de la langue ont perçu très tôt le besoin d'une gestion robuste des erreurs en contexte, et ont rapidement intégré un module de correction dans leurs systèmes. Trois catégories de correction linguistique existent :

1. Approche basée sur la relaxation.
2. Approche basée sur l'expectative.
3. Approche basée sur l'acceptation.

On notera dès à présent que la plupart de ces méthodes ont été pensées pour des domaines restreints, et non pour la correction en contexte de textes non restreints.

Approche basée sur la relaxation. Cette approche estime qu'une erreur est présente dans la phrase dès que son analyse syntaxique échoue (Trawick 1983, Suri 1991, Suri & McCoy 1991). Dans ce cas, le système tâche de relâcher les contraintes syntaxiques exprimées sur la phrase de manière à identifier la ou les règles violées.

Inconvénients. Au vu des limites de l'analyse syntaxique présentée précédemment (cf. Section 13.5.2.1), le module de correction est face à l'alternative suivante :

1. Soit il admet que l'analyse ne couvre pas la langue et accepte d'ignorer de nombreuses alarmes, quitte à laisser passer de vraies erreurs.

2. Soit il pense que l'analyse modélise la totalité de la norme. Il va alors fréquemment déclencher de fausses alarmes, au risque de corriger des cas corrects, mais non ou mal modélisés.

Or, la synthèse de la parole doit pouvoir accepter tous les types de texte, depuis l'œuvre littéraire qui respecte la norme, jusqu'au courrier électronique dont le contenu s'apparente parfois plus à une retranscription de l'oral qu'à un texte rédigé selon la norme. Une analyse linguistique stricte n'est donc pas pertinente, mais une analyse laxiste ne sera pas non plus une solution utile à la synthèse. La synthèse doit en outre pouvoir rapidement intégrer de nouvelles langues. Or, la mise en œuvre de modèles syntaxiques complets pour l'ensemble des langues à couvrir n'est pas envisageable.

Approche basée sur l'expectative. Au cours de l'analyse, cette approche construit la liste des mots qui peuvent apparaître à la position suivante, en fonction de critères syntaxiques et sémantiques, voire pragmatiques (Minton *et al.* 1985, Granger 1983, McCoy 1991). Si le mot suivant du texte n'appartient pas à cette liste, le système suppose avoir localisé une erreur, et tente la correction en choisissant l'un des candidats de la liste.

Inconvénients. Cette approche ne peut se fier qu'à l'analyse morpho-syntaxique, étant donné que l'état de l'art dans les niveaux supérieurs de l'analyse linguistique est encore limité. Or, nous avons montré ci-dessus les limites intrinsèques à l'analyse syntaxique basée sur des règles. La correction à l'aide de cette approche n'est de ce fait réalisable que pour des domaines restreints.

Notons que cette approche s'est par contre montrée pertinente dans d'autres domaines, comme celui de la rédaction assistée, où l'utilisateur se voit proposer une liste de mots suivants en cours de frappe. De nombreux logiciels disponibles sous Linux emploient ce genre de méthodes, comme Open Office ou Kile [7].

Approche basée sur l'acceptation. Cette approche a tendance à privilégier l'information sémantique, et à utiliser peu d'informations des autres niveaux linguistiques (Waltz 1978, Schank *et al.* 1980, Hirst & Budanitsky 2005). La base du système est donc un modèle du discours, par nature réduit à un domaine particulier, étant donné qu'aucun modèle ne permet actuellement de traiter des textes non restreints. Notons cependant qu'une approche récente (Hirst & Budanitsky 2005) a recours au réseau sémantique Wordnet [8], qui modélise une grande partie des concepts pour de nombreuses langues. Wordnet organise principalement les concepts de manière hiérarchique, en déterminant les synonymes, les hyperonymes et les hyponymes.

Dans l'ensemble, l'idée à la base de ces systèmes est la suivante : bien que la plupart des productions textuelles soient remplies d'erreurs, les interlocuteurs ont peu de mal à les

[7] Kile est un éditeur gratuit pour le traitement de texte LaTeX : `kile.sourceforge.net/`.
[8] `wordnet.princeton.edu/`.

interpréter. De ce fait, les erreurs peuvent être ignorées pour autant qu'un sens utile à l'application puisse être trouvé. Dans l'approche de Hirst & Budanitsky (2005), le principe est d'évaluer, dans une fenêtre d'analyse de k mots, la distance entre les différents concepts. Cette distance s'évalue au niveau de l'arborescence de Wordnet. Si un concept semble trop éloigné des concepts qui l'entourent, le système estime avoir détecté une erreur à corriger, et calcule pour ce concept tous les IVs dont la distance d'édition ne dépasse pas 1. Si la distance sémantique de l'un de ces IVs est plus courte que celle de la forme du texte, l'IV en question est accepté comme correction.

Inconvénient. L'approche n'est pas suffisante en synthèse de la parole, parce qu'elle n'apporte qu'une réponse partielle : elle ne s'intéresse qu'aux erreurs sémantiques, alors que la synthèse nécessite la correction de toutes les erreurs *audibles*. Parmi celles-ci, on compte évidemment toutes les erreurs syntaxiques qui entraînent des modifications phonétiques, comme les liaisons (*les oiseaux*).

Analyse

L'analyse linguistique actuelle est encore limitée : rares sont les systèmes qui dépassent l'analyse syntaxique, et ceux qui le font sont limités à un domaine d'application. Le système d'analyse linguistique classique consiste de ce fait en une analyse morpho-syntaxique, l'objectif étant de dégager la structure de la phrase. De ce fait, le module de correction est la plupart du temps dirigé par des informations de nature syntaxique, à partir desquelles la correction est réalisée.

Basée sur une telle structure, la détection et la correction des erreurs n'est pas une tâche triviale :

1. Les couvertures morpho-lexicales et grammaticales sont souvent incomplètes, au vu du temps nécessaire à leur conception. Le système sera donc fréquemment confronté à des *cas limites*, où le choix entre relâchement des contraintes d'analyse et application d'une correction sera difficile.
2. Le système aura souvent des difficultés à localiser l'erreur qui empêche l'analyse de construire la représentation linguistique de la phrase.

Les modèles linguistiques proposés apportent par contre de nombreuses pistes de recherche : la syntaxe de la phrase, les collocations lexicales et la cohésion sémantique sont certainement des indices qui doivent guider le système de correction.

14.5.2 Modèle de langue à orientation lexicale

Cette approche est la première approche statistique apparue dans le domaine de la correction en contexte (Atwell & Elliott 1987, Church & Gale 1991, Mays *et al.* 1991). Elle se fonde sur la métaphore du canal bruité, associé au dactylographe ou au système de reconnaissance de caractères : une phrase X est présentée à l'entrée du canal bruité,

qui y introduit plus ou moins de bruit de sorte que X est transformée en Y à la sortie du canal. L'idée est dès lors de retrouver Y parmi $\{Y\}$, l'ensemble des phrases qui sont des variations obtenues à partir de X, et qui contient X elle-même. Selon la règle de Bayes (cf. Section 13.5.2.2) :

$$Y = \arg\max_{Y} P(X|Y)\, P(Y) \tag{14.5.2.1}$$

Il y a donc deux parties à estimer dans ce modèle : la probabilité que le typographe ait introduit des erreurs, $P(X|Y)$, et la probabilité de la suite lexicale, $P(Y)$.

La probabilité que le typographe ait introduit des erreurs dans la phrase se calcule au niveau du mot :

$$P(X|Y) = \prod_{i=1}^{m} p(x_i|y_i) \tag{14.5.2.2}$$

où m est le nombre de mots de la phrase. Par le théorème de Bayes et en faisant la même approximation que précédemment, ce modèle s'estime :

$$p(x_i|y_i) = p(y_i|x_i)\, p(y_i) \tag{14.5.2.3}$$

où $p(y_i)$ est la probabilité de la correction y_i observée sur un corpus d'entraînement, et $p(y_i|x_i)$ est déterminé en fonction d'un paramètre α. Typiquement, $\alpha = 0,99$. A partir de ce paramètre,

$$p(y_i|x_i) = \begin{cases} \alpha & \text{si } y_i = x_i \\ \frac{1-\alpha}{|\{y^i\}|-1} & \text{sinon} \end{cases} \tag{14.5.2.4}$$

où $\{y^i\}$ est l'ensemble qui comprend le mot x_i et ses corrections, et $|\{y^i\}|$ représente le nombre d'éléments de l'ensemble.

La probabilité de la suite lexicale, $P(Y)$, est estimée comme le produit de la probabilité de tri-grammes lexicaux :

$$P(Y) = \prod_{i=1}^{m} p(y_i|y_{i-2}, y_{i-1}) \tag{14.5.2.5}$$

Il faut noter enfin qu'un ensemble de corrections $\{y^i\}$ n'est construit pour un mot x_i que si la probabilité de son tri-gramme lexical $p(x_i|x_{i-2}, x_{i-1})$ est en-deçà d'un certain seuil. La liste est générée en calculant les mots du dictionnaire qui sont proches en terme de distance d'édition du mot supposé incorrect.

Analyse

Si le modèle est intéressant, il est inapplicable en pratique pour trois raisons :

1. Quelle que soit la taille du corpus utilisé, la majorité des tri-grammes lexicaux n'y apparaît pas ou apparaît peu. La probabilité de la suite lexicale, $P(Y)$, ne peut donc être correctement estimée.
2. Malgré le manque de couverture des corpus, les tri-grammes recensés nécessitent une place de stockage considérable.
3. Le fait que $p(y_i|x_i)$ dépende du nombre d'éléments de $\{y^i\}$ signifie que le poids d'une correction diminue avec le nombre de corrections de l'ensemble. De ce fait, plus l'ensemble contient de corrections, et moins il est probable qu'une correction soit retenue à la place de la forme originale.

14.5.3 Listes de confusion

Cette approche considère que la tâche de correction en contexte est un problème d'ambiguïté : certaines formes lexicales ont tendance à se confondre du fait d'une proximité typographique (*côte* ↔ *côté*, *firme* ↔ *forme*), phonétique (*sceptique* ↔ *septique*) ou sémantique (*entre* ↔ *parmi*).

La base de l'approche consiste dès lors à constituer des listes de confusion, et à déclencher la correction en contexte dès qu'une forme appartenant à une liste est rencontrée. La majorité des systèmes développés dans ce cadre ont fait l'hypothèse que les membres d'une liste de confusion se distinguent principalement en fonction de leur contexte lexical. Deux types de contextes ont été définis : les co-occurrences et les collocations.

Ce point commence par présenter ce que sont les co-occurrences et les collocations, avant de décrire les méthodes proposées pour gérer les listes de confusion :

1. Les listes de décision.
2. Les classificateurs bayésiens.
3. L'approche hybride Tri-Bayes.
4. Le winnowing
5. Les règles apprises.
6. L'analyse sémantique latente.

Une analyse globale qui compare les différents systèmes suit les descriptions.

14.5.3.1 Description des informations contextuelles

Co-occurrences lexicales. Les co-occurrences ont initialement été proposées par Gale *et al.* (1992) dans le cadre de la désambiguïsation sémantique.

Les co-occurrences sont les formes lexicales qui apparaissent fréquemment dans une même fenêtre d'analyse, sans que la position des formes lexicales dans la fenêtre ne soit prise en compte. Dans ce cas-ci, il s'agit des co-occurrences d'une forme w_i appartenant à une liste de confusion, la fenêtre étant de taille $2 * k + 1$ centrée sur w_i.

En ce qui concerne la taille de la fenêtre, Yarowsky (1994) a montré que de petites valeurs ($k \in \{3, 4\}$) sont pertinentes pour résoudre des ambiguïtés syntaxiques locales, tandis que des valeurs relativement élevées ($20 \leq k \leq 50$) sont utiles pour les ambiguïtés sémantiques.

La probabilité de la forme w_i dans ce contexte est formulée de manière bayésienne :

$$p(w_i|w_{i-k}^{i-1},\, w_{i+1}^{i+k}) = \frac{p(w_{i-k}^{i-1},\, w_{i+1}^{i+k}|w_i)\, p(w_i)}{p(w_{i-k}^{i-1},\, w_{i+1}^{i+k})} \tag{14.5.3.1}$$

où (w_{i-k}^{i-1}) note de manière concise la fenêtre gauche $(w_{i-k} \ldots w_{i-1})$, et (w_{i+1}^{i+k}), la fenêtre droite $(w_{i+1} \ldots w_{i+k})$.

Cependant, étant donné que le maximum de vraisemblance $p(w_{i-k}^{i-1},\, w_{i+1}^{i+k}|w_i)$ est par nature difficile à estimer sur un corpus, on fait l'hypothèse que la présence d'une co-occurrence est indépendante de la présence des autres co-occurrences. De ce fait, le maximum de vraisemblance peut être estimé sous la forme d'un produit :

$$p(w_{i-k}^{i-1},\, w_{i+1}^{i+k}|w_i) = \prod_{j \in \{-k \ldots -1,\, 1 \ldots k\}} p(w_j|w_i) \tag{14.5.3.2}$$

dont Gale *et al.* (1994) ont démontré qu'il s'agit d'une approximation raisonnable.

Notons que $p(w_j|w_i)$ reste malgré tout difficile à estimer dans un corpus, raison pour laquelle Golding (1995) propose de poser un seuil T_{min} en-deçà duquel la forme w_j est exclue des co-occurrences. L'exclusion se calcule en fonction du nombre d'occurrences de la forme w_i dans le corpus (C_{w^i}) et du nombre d'occurrences de la forme w_j dans un contexte de $\pm$ k mots autour de w_i (C_{w^j,w^i}). La forme w_j est exclue des co-occurrences pour une liste de confusion L contenant n formes si elle n'apparaît pas suffisamment quelle que soit la forme $w_i \in L$ considérée, ou si au contraire elle apparaît toujours dans le contexte quel que soit la forme $w_i \in L$ considérée :

$$\left.\begin{array}{l} \displaystyle\sum_{1 \leq i \leq n} C_{w^j,w^i} \\ \text{ou} \\ \displaystyle\sum_{1 \leq i \leq n} (C_{w^i} - C_{w^j,w^i}) \end{array}\right\} < T_{min} \rightarrow \text{ exclusion} \tag{14.5.3.3}$$

Les co-occurrences retenues pour une liste de confusion donnée peuvent être ordonnées dans une table de co-occurrences en fonction de leur maximum de vraisemblance.

Collocations. En linguistique, une collocation est une co-occurrence de formes lexicales dont l'arrangement syntagmatique est figé. On s'approche donc des expressions figées, si ce n'est que les termes gardent leur signification pleine. Dans ce cas-ci donc, l'ordre et la position des formes sont pris en compte.

Le principe est dès lors de s'intéresser aux quelques mots qui précèdent ou qui suivent immédiatement la forme w_i appartenant à une liste de confusion donnée, afin de détecter un patron significatif. La Table 14.8 montre par exemple des collocations pour les formes *côte* et *côté*. L'idée est de créer une table de collocations pertinentes pour une liste

vivre	_	*à cote*	→ côte
laisser de	_		→ côté
sur la	_		→ côte
à la	_		→ côte
de tout	_		→ côté
	_	*cœur*	→ côté

TAB. 14.8: Exemples de collocations

de confusion donnée, et de les classer en fonction de leur pertinence. Le classement diffère selon les systèmes (Yarowsky 1994, Golding 1995). Par exemple, Golding (1995) détermine la pertinence d'une collocation c comme sa probabilité maximale d'association avec une forme de la liste de confusion :

$$pert(c) = \max_i p(w_i|c) \tag{14.5.3.4}$$

Certaines collocations peuvent évidemment être réunies selon des critères syntaxiques. De l'exemple de notre Table 14.8, les collocations « *sur la* _ » et « *à la* _ » peuvent être réunies en :

PREP la _ → côte

14.5.3.2 Correction par listes de décision

L'idée est de créer une table hybride et ordonnée de contextes, comprenant à la fois les co-occurrences et les collocations (Yarowsky 1994). L'ordre des contextes de la table dépend de la pertinence calculée sur les co-occurrences et les collocations selon les méthodes décrites précédemment. Dans cette table, co-occurrences et collocations sont donc mélangées.

Correction. Lors du traitement et en présence d'une forme appartenant à une liste de confusion, les différents contextes de la table sont testés dans l'ordre décroissant de leur pertinence. Le processus s'interrompt dès qu'un contexte de la table est trouvé dans la phrase. La forme de la liste de confusion correspondant au contexte sélectionné (co-occurrence ou collocation) est choisie comme correction.

14.5.3.3 Correction par classificateurs bayésiens

Comme dans l'approche précédente, une table hybride et ordonnée comprenant co-occurrences et collocations est construite. L'algorithme de correction est par contre bayé-

sien (Golding 1995).

Correction. En présence dans la phrase d'une forme appartenant à une liste de confusion, l'algorithme pondère les éléments de la liste en fonction de tous les contextes de la table qui apparaissent dans la phrase. Notons que les contextes qui sont des collocations peuvent entrer en conflit : il y a conflit lorsque plusieurs collocations se recouvrent partiellement. Dans ce cas, seule la collocation la plus pertinente est prise en compte. L'algorithme pondère donc les éléments de la liste de confusion en fonction de toutes les co-occurrences, et des collocations qui n'entrent pas en conflit. L'élément de la liste qui a la plus forte probabilité est sélectionné à la fin du processus.

14.5.3.4 Correction hybride « Tri-Bayes »

Cette approche se base sur le constat que les co-occurrences et collocations ne sont pas des informations suffisantes lorsque la distinction entre deux mots d'une liste de confusion est syntaxique (Golding & Schabes 1996). par exemple, « *their*, *there*, *they're* » en anglais, et « *ces*/*ses*, *c'est*/*s'est*/*sais*/*sait* » ou « *son*, *sont* » en français.

L'idée est donc de combiner une analyse syntaxique avec des classificateurs bayésiens. L'analyse syntaxique est ici un modèle de langue.

Modèle de langue syntaxique. Etant donné une phrase $W = w_1 \ldots w_k \ldots w_m$ dont w_k est le mot à corriger et appartient à une liste de confusion L, le système crée pour chaque $w'_k \in L$ une nouvelle phrase W' dans laquelle w'_k est substitué à w_k. Une correction w'_k ne sera préférée à w_k que si $P(W') > P(W)$.

Contrairement au modèle lexical proposé par Mays *et al.* (1991), Golding & Schabes (1996) choisissent ici un modèle syntaxique : pour une phrase W donnée, $P(W)$ est calculé en fonction de la suite de catégories syntaxiques T associées à W :

$$P(W) = P(W|T) \tag{14.5.3.5}$$

Evidemment, certaines formes peuvent accepter plusieurs catégories syntaxiques (par exemple, « *couvent* » peut être un nom ou un verbe). Ceci signifie que $P(W)$ correspond à la suite de catégories qui maximise sa probabilité :

$$P(W) = \max_T \; P(W|T) \tag{14.5.3.6}$$

qui, selon le théorème de Bayes et après simplification, se résout comme suit :

$$\begin{aligned} P(W) &= \max_T \; P(T|W)\, P(T) \\ &= \max_T \prod_{1 \le i \le m} p(w_i|t_i) \prod_{1 \le i \le m} p(t_i|t_{i-2}, t_{i-1}) \end{aligned} \tag{14.5.3.7}$$

où $p(t_i|t_{i-2}, t_{i-1})$ est un tri-gramme syntaxique beaucoup plus facile à estimer à partir d'un corpus étiqueté syntaxiquement que ne l'est le tri-gramme lexical de Mays *et al.* (1991). Notons que Golding & Schabes (1996) ne déterminent pas comment $p(w_i|t_i)$ est estimé, alors que, comme nous l'avons expliqué en Section 13.5.2.2, il est difficile d'estimer la probabilité de toutes les associations {forme, catégorie} à partir d'un corpus. De même, la méthode de lissage de $p(t_i|t_{i-2}, t_{i-1})$ n'est pas détaillée.

Ce modèle de langue seul est très performant pour choisir une correction lorsque les formes d'une liste s'opposent au niveau de leurs catégories. Il est par contre inopérant lorsque les formes ont la même analyse syntaxique : les formes sont dans ce cas exclusivement départagées en fonction de leur probabilité d'association avec la catégorie syntaxique : $p(w_i|t_i)$. Ceci explique que les auteurs aient décidé de le combiner avec des classificateurs bayésiens, qui ont montré leur pertinence au niveau des formes de même catégorie syntaxique.

Correction. Les deux méthodes ne sont pas combinées dans l'évaluation de la bonne correction. Seule l'une de ces méthodes est employée en fonction de la forme à corriger. Si la forme appartient à une liste de confusion dont les membres diffèrent au niveau de leur catégorie syntaxique, le modèle de langue est utilisé. Si la forme appartient à une liste dont les membres partagent la même analyse syntaxique, le classificateur bayésien est employé.

Les auteurs ne précisent pas comment ils gèrent les listes dont une partie des éléments seulement diffère au niveau des catégories. Or, on peut penser que dans ce cas de figure, il serait bon de combiner les deux méthodes dans la prise de décision.

14.5.3.5 Correction par winnowing

Le verbe anglais *winnow* correspond à l'expression française « séparer le bon grain de l'ivraie ». L'objectif de l'algorithme initialement développé par littlestone (1988) est d'aider la prise de décision ou la classification d'un élément en sélectionnant automatiquement les meilleures caractéristiques parmi un ensemble de caractéristiques disponibles.

Ce principe a été repris par Golding & Roth (1996, 1999) dans le cadre de la correction en contexte. L'idée générale de l'algorithme est de déterminer par un apprentissage les caractéristiques les plus pertinentes parmi les caractéristiques disponibles pour une forme appartenant à une liste de confusion donnée. Seules les caractéristiques pertinentes seront ensuite utilisées lors d'une prise de décision. L'entraînement se divise donc en deux étapes :

1. Génération des caractéristiques « disponibles ».
2. Apprentissage des caractéristiques pertinentes.

Caractéristiques disponibles. Pour une forme donnée appartenant à une liste de confusion, les caractéristiques sont comme précédemment ses co-occurrences et ses collocations. La liste des caractéristiques disponibles est construite comme dans l'approche bayésienne en relevant, dans les phrases d'un corpus, les contextes de la forme qui sont les plus fréquents.

Réseau. C'est à partir de ce point que le modèle diffère de l'approche bayésienne. Les auteurs ont recours à un réseau de nœuds de type neuronal, tel que décrit dans (Valiant 2004) et illustré par la Figure 14.6.

Le principe de base est de créer un nœud *conceptuel* pour chaque forme d'une liste de confusion et un nœud *contextuel* pour chaque caractéristique contextuelle retenue par la phase précédente. Un nœud conceptuel est relié par une transition pondérée aux nœuds contextuels qui ont été rencontrés dans les mêmes phrases. Lors du traitement d'une phrase donnée, les nœuds contextuels trouvés dans la phrase sont activés, et les nœuds conceptuels qui y sont reliés produisent une sortie binaire : 0 ou 1. Un nœud ne produit 1 que si la somme des pondérations des nœuds contextuels qui lui sont reliés et qui ont été activés est supérieure à un seuil θ donné :

$$1 \text{ ssi } \left(\sum_i w_i\right) > \theta \tag{14.5.3.8}$$

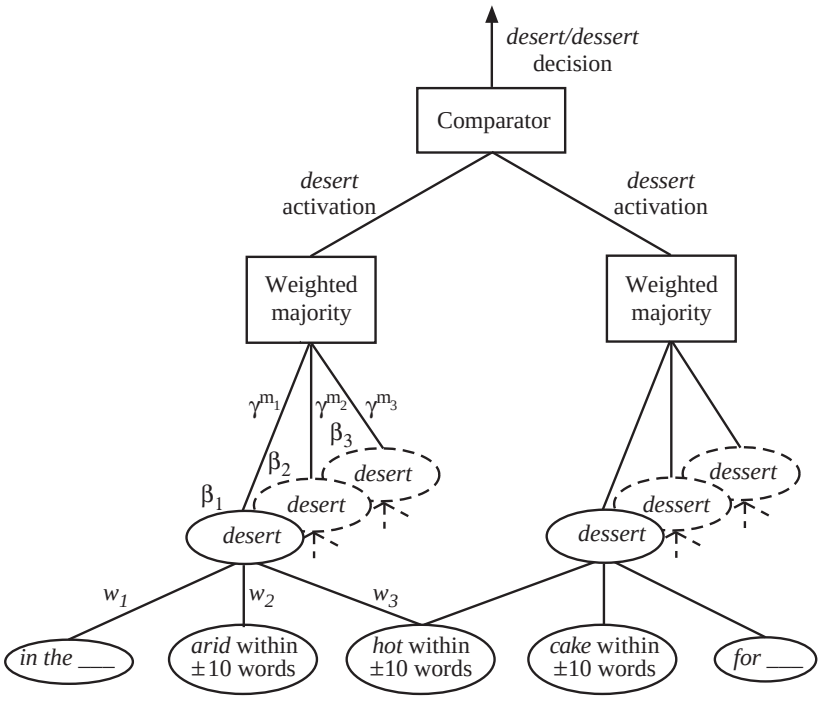


FIG. 14.6: Approche winnow. Figure reprise de (Golding & Roth 1999). Les nœuds inférieurs du réseau sont les caractéristiques contextuelles (co-occurrences et collocations). Les « nuages » intermédiaires correspondent aux formes de la liste de confusion {*desert*, *dessert*}. La partie supérieure du système décide de la forme à activer en fonction du principe de « majorité pondérée »

Apprentissage. Le mode général d'apprentissage utilisé a été proposé par Valiant (1995) et permet une adaptation du réseau en cours d'utilisation.

Winnow. Un même poids w est attribué au départ à toutes les transitions reliées à un concept donné. En cours d'apprentissage pour une phrase donnée, on compare le résultat (0 ou 1) produit par un nœud conceptuel au résultat attendu. Si le résultat attendu est positif (1) et que le résultat produit est différent (0), les poids des transitions activées sont revus à la hausse à l'aide d'un facteur α supérieur à 1. Si au contraire le résultat attendu est négatif (0) et que le résultat produit est différent (1), les poids des transitions activées sont revus à la baisse à l'aide d'un facteur β inférieur à 1. Au cours de l'apprentissage, les poids des différentes transitions évoluent différemment, de sorte que certaines transitions vont complètement disparaître alors que d'autres vont être favorisées. C'est donc ici qu'est appliqué le principe *winnow*.

Majorité pondérée. Les auteurs ont estimé qu'un seul nœud par concept n'est pas suffisant, et ont eu recours à un algorithme de décision par majorité pondérée (Littlestone & Warmuth 1994). Pour chaque concept, un nuage de 5 nœuds conceptuels est créé. Chacun de ces nœuds conceptuels est évidemment relié aux mêmes nœuds contextuels, mais est repondéré différemment lorsqu'une erreur est observée : le facteur β varie de 0,5 pour le premier nœud à 0,9 pour le cinquième nœud. En fonction de ces repondérations différentes, les nœuds conceptuels peuvent différer dans leurs sorties. Ceci permet de déterminer un classement des nœuds du nuage, en tenant compte du nombre d'erreurs commises par chaque nœud. Ce classement est représenté dans la Figure 14.6 par le facteur γ qui relie chaque nœud du nuage à un système de prise de décision pondérée où chaque nœud du nuage intervient en fonction de la fiabilité qui lui a été accordée.

Correction. Lors du traitement d'une phrase, si la phrase contient une forme d'une liste de confusion, les nœuds contextuels trouvés dans la phrase sont activés, et les nœuds des nuages conceptuels qui y sont reliés produisent une sortie binaire : 0 ou 1. Les sorties des différents nœuds d'un même nuage sont combinées par majorité pondérée. La forme qui reçoit le poids global le plus élevé est émise par le réseau. Le réseau fournit donc une seule correction, et non une liste ordonnée de corrections.

14.5.3.6 Correction par règles apprises

Mangu & Brill (1997) ne contestent certainement pas les très bons résultats des systèmes précédents, mais estiment que leurs approches statistiques manquent de possibilité d'ajustement manuel : les paramètres sont pondérés par l'apprentissage, ce qui rend difficile une interprétation du comportement du système. Les auteurs reconnaissent par contre l'intérêt d'un apprentissage automatique, propice à l'adaptation rapide du système dans de nouvelles langues.

Principe. L'approche concerne les listes de confusion et se fonde sur la description de règles de transformation du type « A devient B lorsque le contexte contient X dans les k mots ». Ces règles ne sont cependant pas construites manuellement par des experts, mais construites à partir d'une phase d'apprentissage qui détermine, comme dans les autres méthodes, les contextes (co-occurrences et collocations) les plus fréquents pour les formes d'une liste de confusion donnée. A ce stade, les règles ne sont donc qu'une transposition des contextes des méthodes précédentes dans un formalisme compréhensible par l'être humain.

Apprentissage. Lorsque les règles ont été construites, une phase d'entraînement est réalisée selon l'algorithme proposé par Brill (1995), dont le principe est le suivant :

1. On pose $L = \emptyset$, une liste de transformations.
2. Une copie C_0 du corpus d'apprentissage A subit une phase de neutralisation où chaque forme d'une liste de confusion est remplacée par la forme la plus fréquente de la liste.
3. Tant que le nombre de différences entre A et C_0 est supérieur à un seuil donné,
   (a) Pour chaque règle de transformation R_i possible,
      i. Une copie C_i est créée à partir de C_0.
      ii. R_i est appliquée à C_i.
      iii. Le nombre de différences entre A et C_i est mémorisé.
   (b) La règle R_i qui minimise le nombre d'erreurs est ajoutée à la liste L, et est appliquée à C_0.

La description de l'apprentissage montre que la même règle peut être ajoutée plusieurs fois dans la liste. Les auteurs notent que ce principe permet aux règles de se corriger mutuellement.

Correction. Lorsqu'une forme appartenant à une liste de confusion est identifiée par le système, elle est remplacée par la forme la plus probable de la liste de confusion, et toutes les règles de la liste de transformations qui lui sont applicables lui sont appliquées. Ceci signifie que la forme est susceptible d'être modifiée plusieurs fois au cours de l'application des règles.

Les listes de décision n'appliquaient qu'un seul contexte, le plus probable. Les approches bayésiennes combinaient l'ensemble des contextes correspondants. Cette nouvelle approche n'utilise qu'une partie des contextes et les applique dans un ordre précisé au cours de l'apprentissage.

Remarque. Les auteurs constatent que la méthode présente deux avantages par rapport au winnowing : le nombre de règles retenues dans la liste est nettement inférieur au nombre de contextes utilisés par winnow, et la liste de transformations, tout à fait lisible, peut être modifiée au besoin.

14.5.3.7 Correction par analyse sémantique latente

L'analyse sémantique latente, ou LSA [9], a initialement été développée dans le cadre de la recherche d'information, domaine dans lequel on parle également d'indexation sémantique latente, ou LSI [10] (Dumais *et al.* 1988, Deerwester *et al.* 1990).

Principe. La LSA part de l'hypothèse que l'auteur a une idée ou une information à communiquer, dont les formes lexicales du texte sont des *témoins*. L'objectif est dès lors de découvrir l'idée ou l'information derrière les formes. Cependant, étant donné que de nombreuses formes sont polysémiques ou synonymes, les témoins du texte sont quelque peu bruités ou ambigus.

L'objectif de la LSA est d'éliminer ce bruit. Le principe est de représenter le texte dans un espace multidimensionnel, et de réduire ensuite cet espace à ses dimensions les plus importantes. A ce stade, le bruit a disparu du texte, et une interprétation sémantique devient possible.

A partir d'une collection de textes, on crée un espace multidimensionnel sous la forme d'une matrice M de taille $m \times n$ dont les lignes sont les m formes lexicales de la collection et dont les colonnes sont les n documents de la collection. Une cellule i, j donnée contient une valeur qui est une fonction de la fréquence de la forme i dans le document j et dans l'ensemble de la collection. A ce stade, le contenu est encore bruité.

Une décomposition en valeurs singulières (SVD [11]) est ensuite réalisée sur la matrice. Nous laissons le lecteur intéressé par les détails de la SVD se reporter à (Eckart & Young 1936) pour une explication détaillée. Le résultat de la SVD est une matrice M' de taille $m \times n$ qui ne considère que les k premières dimensions de M et dont le bruit a disparu. En somme, M' représente ou plutôt prédit la fréquence avec laquelle chaque forme lexicale devrait apparaître dans un document selon son contenu sémantique. Bien sûr, cette prédiction n'est possible que pour les domaines sémantiques couverts par la matrice initiale.

La SVD étant réalisée, la similarité sémantique d'un nouveau texte par rapport à la collection initiale peut être déterminée. L'estimation de la similarité sémantique entre deux documents est réalisée en calculant le cosinus de l'angle formé par leurs vecteurs respectifs. Pour déterminer le document de la collection le plus proche du nouveau texte, on calcule donc un vecteur de fréquences pondérées des formes lexicales du nouveau texte, que l'on projette dans l'espace multidimensionnel de M'.

L'application de la LSA en correction contextuelle a été proposée par Jones & Martin (1997).

Entraînement. Une matrice M doit être créée pour chaque liste de confusion prise en compte dans la correction. Dans ce processus, chaque phrase du corpus source est traitée

[9] *Latent Semantic Analysis.*
[10] *Latent Semantic Indexing.*
[11] *Singular Value Decomposition.*

ici comme un document et n'intègre la matrice que si elle contient une forme de la liste de confusion. Les phrases intégrées subissent quelques transformations :

1. Réduction : la phrase est réduite à une fenêtre de $2k+1$ centrée sur la forme lexicale.
2. Lemmatisation : les formes de la phrase sont remplacées par leurs formes canoniques. Par exemple, *couvent* sera représenté par *couver* ou *couvent* selon l'analyse syntaxique.
3. Bi-grammes : pour les mots conservés, des lignes supplémentaires sont ajoutées et contiennent des bi-grammes de mots.

Correction. Lorsqu'une phrase p_0 contient une forme f_0 appartenant à une liste de confusion L,

1. La matrice M'_L correspondante est sélectionnée.
2. Pour chaque forme f_i de la liste de confusion,
   (a) Une copie de la phrase p_i est réalisée, dans laquelle f_i est substituée à f_0.
   (b) Les mêmes transformations que celles réalisées sur les phrases de la matrice (réduction, lemmatisation, bi-grammes) sont appliquées à p_i.
   (c) f_i est retirée de p_i, pour ne pas biaiser la comparaison ultérieure. Seuls les bi-grammes en gardent donc la trace.
   (d) Un vecteur v_i est créé à partir de p_i et projeté dans l'espace de la LSA. La forme de confusion f_j sélectionnée est celle dont le vecteur forme le cosinus c_j le plus important avec v_i. Le processus retourne la paire $\{f_j, c_j\}$.
3. Parmi toutes les paires $\{f_j, c_j\}$, on retient celle qui a le cosinus le plus important. La forme associée est considérée comme la correction la plus probable.

14.5.3.8 Analyse des résultats

La Table 14.9 présente un condensé des résultats fournis et des comparaisons réalisées par certains auteurs (Golding 1995, Golding & Schabes 1996, Golding & Roth 1996, Mangu & Brill 1997, Jones & Martin 1997). Tous les résultats présentés ont été obtenus sur les mêmes corpus, utilisés à la fois pour l'entraînement et pour le test : le corpus Brown contenant 1 million de mots (Kučera & Francis 1967) et un corpus d'articles du Wall Street Journal contenant 750 000 mots (Marcus *et al.* 1993).

La table présente deux groupes d'évaluations (1 et 2), parce que les entraînements et les tests n'ont pas été réalisés sur les mêmes parties des corpus. Les deux évaluations partagent cependant la méthode bayésienne et sont donc « comparables ».

La table contient deux groupes de listes de confusion (I et II). Les listes du premier groupe, commençant par (*affect*, *effect*) contiennent des formes de même catégorie grammaticale, contrairement à celles du second groupe, commençant par (*accept*, *except*).

Les systèmes proposaient parfois plusieurs résultats, faisant varier certains paramètres du système. Dans un soucis de lisibilité, nous ne présentons que le meilleur résultat de chaque système.

		(1)		(2)				
		List. Déc.	**Bayes**	**Bayes**	**Tri-Bayes**	**Winnow**	**Règles**	**LSA**
(I)	*affect*, *effect*	82,1	82,7	95,9	95,9	**100,0**	**100,0**	94,3
	among, *between*	65,9	78,6	75,3	75,3	**89,2**	84,4	80,8
	amount, *number*	62,9	66,2	82,9	82,9	85,4	87,8	**88,4**
	country, *county*			85,5	85,5	**96,8**	**96,8**	81,3
	peace, *piece*	85,2	85,2	**90,0**	**90,0**	88,0	**90,0**	83,9
	principal, *principle*	81,2	81,2	85,3	88,2	**91,2**	**91,2**	**91,2**
	raise, *rise*	80,4	80,7	74,4	76,9	**89,7**	84,6	80,6
(II)	*accept*, *except*	78,9	81,1	88,0	82,0	**96,0**	92,0	82,3
	begin, *being*	84,2	86,9	91,8	97,3	**99,3**	97,9	93,2
	lead, *led*	84,0	84,0	79,6	83,7	**93,9**	**93,9**	73,0
	passed, *past*	93,2	92,4	89,2	**95,9**	93,2	90,5	80,3
	quiet, *quite*			89,4	**95,5**	93,9	93,9	90,8
	than, *then*	96,7	97,3	93,2	94,9	**96,7**	96,1	90,5
	weather, *whether*	93,5	93,5	96,7	93,4	**100,0**	**100,0**	85,1
	Moyenne	82,4	84,2	86,9	88,4	**93,8**	92,8	85,4
	Rang	5-6	4		3	**1**	2	5-6

TAB. 14.9: Listes de confusion : évaluation

Les résultats. La meilleure méthode est incontestablement l'approche winnow, suivie de près par l'approche par règles.

La LSA et les listes de décision proposent les moins bons résultats. Les moins bonnes performances de la LSA sont probablement dues au fait que les formes de la phrase sont lemmatisées. Or, au-delà des liens sémantiques, la correction doit également tenir compte de la configuration syntaxique de la phrase, information qui n'est prise en compte d'aucune manière par la LSA. Les listes de décision, quant à elles, souffrent certainement du fait qu'elles ne considèrent que le meilleur contexte de la liste, alors que les autres méthodes prennent toutes en compte un ensemble plus ou moins important de contextes pour la correction.

Le classement en milieu de peloton de l'approche hybride Tri-Bayes est décevant, si l'on considère que ce système est le seul à prendre en compte de l'information syntaxique. Cependant, l'information est éminemment *locale* :

1. Le tri-gramme, par nature, ne tient pas compte de dépendances de longue distance.
2. Toutes les hypothèses syntaxiques ne sont pas évaluées en même temps : une seule catégorie par mot est utilisée dans le calcul de la probabilité d'une phrase.
3. Le modèle souffre probablement de cette dichotomie entre information syntaxique et

lexicale. Au lieu de ne tenir compte que de l'une des deux sources d'information dans l'estimation de la pertinence d'une correction, le mieux serait probablement de les combiner.

Analyse globale. De manière générale, toutes ces approches présentent les mêmes inconvénients :

1. Elles ne sont applicables qu'au cas des listes de confusion. Une méthode plus globale est souhaitable.
2. Les méthodes nécessitent un entraînement pour pouvoir gérer une liste de confusion donnée. L'entraînement doit donc couvrir *toutes* les listes possibles.
3. Les listes de confusion sont préétablies. Or, il pourrait être intéressant que le système soit capable de générer *au vol* des confusions possibles pour un mot donné.

14.5.4 Extraction de caractéristiques multi-niveaux

L'approche proposée par Schaback & Li (2007) est utilisée dans un système de correction interactive qui gère à la fois les OOVs et les erreurs sur IVs.

Informations utilisées. Le principe est de gérer les deux types d'erreurs selon un seul et même canevas. Qu'il s'agisse d'un IV ou d'un OOV, le mot traverse les étapes suivantes :

1. Génération des candidats :
   (a) Génération des formes de prononciation phonétique similaire, par clef de similarité (SOUNDEX).
   (b) Génération des formes proches par application de la distance d'édition décrite dans (Brill & Moore 2000).
   (c) Classement et sélection des 5 meilleures solutions.
2. Pour chaque candidat de la liste, le système calcule :
   (a) Deux bi-grammes lexicaux : $p(w_i|w_{i-1})$ et $p(w_{i+1}|w_i)$. Les auteurs assurent que le bi-gramme est une fenêtre suffisamment petite pour permettre une estimation correcte sur un corpus de grande taille.
   (b) Des collocations, dont l'estimation diffère de la méthode classique : le terme concerné est lemmatisé (*bought* $\rightarrow$ *buy*), tandis que les mots du contexte sont remplacés par leurs catégories grammaticales. Les trois collocations suivantes sont estimées : $p(w_i|t_{i-2}, t_{i-1})$, $p(w_i|t_{i-1}, t_{i+1})$ et $p(w_i|t_{i+1}, t_{i+2})$, où t_i est la catégorie grammaticale du mot w_i. Le système se situe donc entre les tri-grammes syntaxiques classiques et les collocations.
   (c) Des co-occurrences, dans une fenêtre de maximum 10 mots. Notons que les 10 mots sont comptés après que les termes appartenant à une *Stop List* aient été supprimés.

Les différentes données calculées (cf. Table 14.10) sont représentées sous la forme de vecteurs $\overrightarrow{X_i}$ de nombres réels normalisés entre 0 et 1.

Bi-gramme gauche
Bi-gramme droit
Fréquence du candidat
Produit des bi-grammes
Collocation gauche
Collocation centre
Collocation droite
Produit des collocations
Co-occurrences

TAB. 14.10: Vecteur de données (probabilités ou fréquences normalisées)

Modèle de correction. Les auteurs utilisent une machine à support vectoriel (SVM [12]) pour réaliser la classification d'un vecteur donné. Pour une description détaillée de ce qu'est un SVM, nous renvoyons le lecteur vers le tutoriel (Cortes & Vapnik 1995) ou l'article détaillé (Burges 1998). Dans le principe, un SVM est un classificateur linéaire ou non linéaire qui présente l'avantage de maximiser l'écart entre les hyperplans de l'espace euclidien de représentation des données (cf. Figure 14.7). Il est construit par une phase d'entraînement où chaque vecteur de données est accompagné de sa classification. La classification est initialement binaire $(-1, 1)$. En phase de prédiction, le SVM donne cependant une estimation plus ou moins proche de l'une ou l'autre classe de l'apprentissage.

Pour la correction, le SVM est entraîné sur un ensemble de triplets (candidat, vecteur, classe) à partir d'un corpus d'erreurs présentées avec leurs corrections. Pour chaque erreur, 5 triplets sont générés, 1 par candidat à la correction. La classe vaut 1 pour le candidat correct, et -1 pour les autres.

En phase de correction, 5 couples (candidat, vecteur) sont créés, et le SVM retourne des valeurs situées entre les deux classes -1 et 1. Le candidat le plus proche de 1 est choisi comme correction.

Analyse. L'approche semble donner de bons résultats, bien que la comparaison réalisée avec d'autres systèmes ne soit pas aisée à analyser. L'avantage majeur de cette approche est la gestion simultanée des OOVs et des IVs. Cependant, dans l'ensemble, les inconvénients suivants semblent se dégager :

1. En ce qui concerne la génération des candidats,

[12] *Support Vector Machine.*

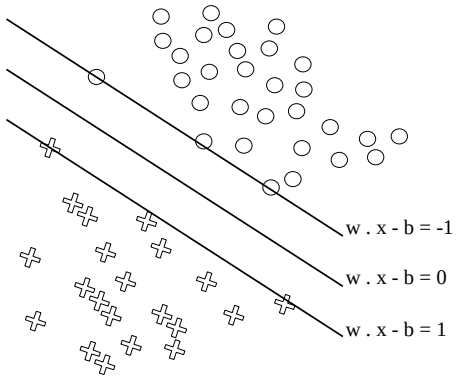


FIG. 14.7: SVM : l'écart entre les hyperplans des deux classes est maximal. Les données situées le long des hyperplans constituent les vecteurs supports

   (a) Il est dommage que le même nombre d'erreurs soit autorisé quelle que soit la longueur du mot.

   (b) Il serait probablement intéressant de différencier la génération des candidats dans le cas des IVs et des OOVs.

2. Les collocations utilisent des formes lemmatisées. Or, il est évident que les diverses formes d'un même lemme ne se rencontrent pas forcément dans les mêmes collocations. La méthode a donc tendance à introduire un certain degré de confusion au niveau syntaxique.
3. Tel qu'il est décrit, le modèle ne semble pas autoriser la gestion de plusieurs analyses grammaticales pour la même forme. On peut supposer que seule la catégorie la plus probable intervient dans le calcul des différentes valeurs du vecteur.
4. De manière générale, et bien que des informations de plusieurs niveaux soient utilisées, le système continue à prendre une décision au niveau *local*. Aucun modèle ne tente de valider une solution au niveau de la phrase. Or, le classement du SVM pourrait être intégré dans une prise de décision plus large.

14.5.5 Exploitation du Web

De nos jours, le Web est incontestablement devenu la plus grande source de documents gratuitement et facilement consultables. De nombreuses études se sont de ce fait orientées vers l'exploitation de cette source particulière. Le Web a ainsi été utilisé comme corpus dans plusieurs approches linguistiques (Volk 2001, Kilgarriff & Grefenstette 2003).

En correction, certains auteurs ont amélioré l'estimation de n-grammes absents de leurs corpus en intégrant dans leurs estimations les requêtes effectuées sur les moteurs de recherche (Zhu & Rosenfeld 2001, Keller & Lapata 2003).

D'autres utilisent le Web pour constituer des corpus d'entraînement dépendant du domaine (Strohmaier *et al.* 2003, Baroni & Bernardini 2004, Williams & Zobel 2005, Ringlstetter *et al.* 2007). L'idée est ici de choisir le modèle de langue à appliquer à un texte en fonction du domaine auquel il appartient. Ceci implique, bien sûr, de disposer d'une méthode permettant de déterminer le domaine d'un texte avant de choisir le modèle de langue à lui appliquer.

Dans l'ensemble, il s'agit donc d'approches statistiques relativement classiques, si ce n'est qu'elles utilisent un corpus d'entraînement particulier.

De la littérature que nous avons parcourue, une approche se distingue particulièrement. Cucerzan & Brill (2004) proposent un modèle destiné à la correction des requêtes réalisées sur le Web. Ils constatent que les requêtes sont en général composées d'un ou de plusieurs concepts exprimés à l'aide de mots absents des dictionnaires. Ils en concluent que les méthodes habituellement appliquées en correction isolée ou contextuelle ne sont pas applicables à la correction des requêtes : d'une part, la vérification de l'existence des mots dans un lexique est inutile, et d'autre part, les requêtes, trop courtes et sans véritable structure grammaticale, n'autorisent aucune analyse syntaxique, fût-elle statistique.

L'approche proposée est de logger les requêtes précédemment formulées afin de s'en servir comme base de correction pour les nouvelles requêtes. Le principe est de déterminer une distance d'édition sous la forme d'un modèle de langue, sans référence au lexique :

Etant donné $X \in \Sigma^*$*, trouver* $Y \in R$ *tel que :*

$$Y = \max_{Y \in R : D(X,Y) \leq \delta} P(Y|X)$$

où R est le log des requêtes précédentes. Ce principe n'est cependant pas applicable à la correction en synthèse, où les textes présentent fréquemment une structure syntaxique consistante et des termes appartenant au lexique. L'abondance et la longueur des textes en synthèse ne permettent pas de concevoir un log de ce type.

Une caractéristique de la méthode est cependant intéressante. La distance d'édition est appliquée de manière itérative, de sorte que des corrections de proche en proche permettent d'arriver progressivement à une requête corrigée, en passant par des requêtes, mémorisées dans les logs, comportant de moins en moins d'erreurs :

anol scwartegger
↓
*a**r**nol**d** sc**h**wart**n**egger*
↓
*arnold schwar**z**negger*
↓
*arnold schwarz**e**negger*

14.5.6 Synthèse des approches présentées

Les premières méthodes proposées en correction dépendante du contexte étaient linguistiques. Elles se répartissent principalement entre méthodes syntaxiques et méthodes sémantiques. Si les premières butent principalement sur les problèmes de rupture syntaxique et sur la difficulté de proposer un modèle capable de tenir compte de toutes les structures valides, les secondes ne sont actuellement applicables que dans des domaines spécifiques. Dans l'ensemble cependant, la véritable limite de l'approche linguistique est la difficulté de l'adapter à différentes langues.

Pour dépasser ces limites, les chercheurs se sont tournés vers les méthodes statistiques. Les premiers systèmes proposés se sont basés sur des modèles de langue à orientation lexicale. Cependant, la méthode a buté sur la difficulté à proposer une estimation consistante de l'ensemble des suites lexicales possibles.

Ceci a conduit les chercheurs à limiter leurs travaux aux seules listes de confusion. De très nombreuses méthodes ont été proposées dans ce domaine particulier de la correction dépendante du contexte, et produisent des résultats très satisfaisants. Que la méthode soit linguistique ou statistique, l'entraînement du modèle ou la construction des règles nécessitent cependant la constitution préalable de listes de confusion. De ce fait, toute confusion absente de la liste ne peut être gérée. Il serait intéressant de pouvoir dépasser cette limite.

Une approche se distingue considérablement des précédentes. Son principe est de tenir compte d'informations extraites de plusieurs niveaux linguistiques afin de corriger, dans un même élan, les OOVs et les erreurs dépendantes du contexte. Le niveau syntaxique est cependant utilisé de manière fort locale : la méthode s'intéresse à la probabilité d'un n-gramme hors de tout contexte. Il nous semble en outre qu'il pourrait être intéressant de spécialiser les informations utilisées selon le type de l'erreur.

Les systèmes les plus récents exploitent massivement les ressources du Web pour améliorer les performances de leurs modèles de langue. Ces approches produisent de très bons résultats, mais nécessitent de déterminer le domaine auquel appartient le document à corriger. Parmi les méthodes exploitant le Web, certaines se concentrent sur la correction des requêtes réalisées à l'aide de moteurs de recherche. Si ces méthodes sont encore contextuelles, elles s'écartent cependant de la notion de *phrase*, pour ne considérer que des suites de mots-clefs, sans structure syntaxique. Dans ce domaine, la correction consiste principalement en une comparaison entre la requête et un historique des requêtes précédentes.

14.6 Conclusion

14.6.1 Typologie des erreurs

Les premiers travaux en correction orthographique ont vu le jour dans les années '60 et les recherches dans le domaine sont encore nombreuses aujourd'hui. L'état de l'art que nous avons constitué ne peut dès lors être que parcellaire. Nous pensons, cependant, avoir

recensé la grande majorité des tendances du domaine.

Dans l'ensemble, les divers travaux de recherche se sont répartis entre détection et correction des non-mots, et correction des erreurs dépendantes du contexte.

1. Les recherches dans le domaine de la détection ont montré que l'utilisation d'un dictionnaire n'offre aucune certitude. Cependant, l'utilisation de modèles n-grammes seuls oblige le système à avancer à l'aveuglette, en s'aidant d'un seuil de détection, arbitraire par nature. Le dictionnaire, malgré son caractère incomplet, paraît donc préférable, parce qu'il a au moins l'avantage d'exclure de la détection les termes qui lui appartiennent. Il ne faut cependant pas oublier qu'un terme du dictionnaire peut être erroné dans le contexte...
2. La correction des OOVs n'est pas un problème résolu, mais a donné lieu à de nombreuses techniques très performantes. La distance d'édition, qui paraissait un peu en retrait par rapport aux autres méthodes que sont les clefs de similarité, les systèmes par règles, les n-grammes et les réseaux de neurones, a montré qu'elle pouvait produire des résultats certainement équivalents, pour autant que la modélisation de la distance soit un peu plus fine que les simples opérations d'édition proposées initialement. Aucune méthode n'est donc *a priori* à exclure.
3. 25 à 50% des erreurs relevées dans les textes sont dépendantes du contexte. Les corriger est donc une nécessité. De nombreuses techniques ont ainsi été conçues afin de gérer au mieux ce type d'erreurs. L'analyse de l'état de l'art ne permet cependant pas de favoriser une approche particulière, tant les corrections envisagées sont différentes. La critique principale qui décourage la mise en place d'une approche purement linguistique est la difficulté à adapter le système à plusieurs langues. Sans cette limite, l'approche serait certainement la plus pertinente. En effet, les méthodes statistiques n'apportent pas *la* solution : les modèles lexicaux sont difficiles à estimer, les listes de confusion réduisent le problème à quelques paires ciblées, et l'exploitation du Web apporte principalement une meilleure estimation du modèle de langue, mais implique la possibilité de déterminer le domaine d'application du texte à corriger. Dans l'ensemble, aucune des approches statistiques proposées ne répond au problème particulier de la synthèse, qui est de corriger les erreurs dépendantes du contexte qui sont audibles dans le flux de parole.

L'état de l'art nous fournit donc une base solide concernant la détection et la correction des non-mots. Par contre, les recherches réalisées dans le domaine de la correction dépendante du contexte semblent répondre à des besoins qui s'éloignent fortement de ceux de la synthèse.

Dès lors, que pouvons-nous dégager de l'état de l'art en correction dépendante du contexte ?

1. La correction contextuelle concerne la phrase et doit se situer à ce niveau. La prise de décision locale n'est pas pertinente.

2. Les modèles statistiques peuvent être employés en correction contextuelle. Il serait cependant probablement intéressant de les combiner à des méthodes plus linguistiques.
3. Une correction progressive, comme celle proposée par Cucerzan & Brill (2004) dans son système de correction de requêtes, est pertinente. Nous en déduisons qu'un non-mot qui présente une erreur contextuelle pourrait être corrigé en deux étapes. En voici un exemple :

ils él<u>eb</u>ve	(OOV)
↓	
*ils él<u>**è**</u>v<u>e</u>*	(IV, mais erreur d'accord)
↓	
*ils élèv<u>**ent**</u>*	(IV, correctement accordé)

14.6.2 Machines à états finis

Pendant longtemps, les rares systèmes de correction qui ont eu recours aux machines à états finis ne les ont employées que pour représenter leurs dictionnaires. En somme, les machines étaient exclusivement vues comme un moyen de *compresser l'information*.

Leur champ d'application s'est cependant récemment élargi, lorsqu'Oflazer (1996) les a utilisées pour simuler les opérations de la distance d'édition. D'autres chercheurs ont ensuite suivi la même voie. Ces méthodes ont certainement eu le mérite de montrer l'intérêt des machines à états finis, qui factorisent les opérations communes aux termes qui partagent les mêmes séquences de symboles, et permettent de ce fait la consultation rapide de dictionnaires contenant plusieurs centaines de milliers de formes.

Cependant, les deux constats suivants doivent être dressés :

1. Les machines à états finis n'ont été employées que pour modéliser la distance d'édition. En outre, les modèles proposés restent fort proches de la distance d'édition classique : le nombre d'erreurs ne varie pas en fonction de la longueur du mot, les opérations ne sont pas contextualisées et les conversions phonétiques ne sont pas prises en charge.
2. Les machines ne sont qu'un outil parmi tant d'autres. Avant et après la distance d'édition, le système manipule des strings et recourt à d'autres modèles. En somme, aucun profit n'est tiré des nombreuses opérations régulières définies sur les machines à états finis.

En correction orthographique, une approche exclusivement basée sur des machines à états finis n'a donc pas encore été proposée.

Chapitre 15

Intégration de la correction orthographique en synthèse

L'intégration d'un système de correction dans notre synthétiseur repose sur un certain nombre de postulats, que nous présentons dans la Section 15.1. Nous en analysons ensuite, en Section 15.2, l'impact sur la typologie des erreurs prises en compte et sur l'architecture de l'analyseur linguistique, dont les modules morphologique et syntaxique ont dû être fusionnés.

Ceci étant posé, nous présentons, en Section 15.3, l'algorithme d'analyse morpho-syntaxique dans son ensemble. Nous en donnons tout d'abord un aperçu global, avant d'en détailler les grandes phases et de mettre l'accent sur les endroits où la correction orthographique prend place.

Sur cette base, nous attirons l'attention, en Section 15.4, sur ce que nous appelons les *lignes de faîte* de l'algorithme : l'interface de communication entre les processus, et quelques méthodes spéciales de gestion des machines à états finis qui sous-tendent tout l'algorithme. La Section 15.5 présente ensuite les différents modèles dédiés à l'analyse morpho-syntaxique. L'ensemble du processus standard étant établi, nous nous concentrons, en Section 15.6 sur les modèles dédiés à la correction orthographique.

Le chapitre se referme en Section 15.7 par une évaluation du système développé. Les conclusions que nous en dégageons sont présentées au Chapitre 17, afin d'inclure dans une même réflexion l'approche que nous proposons au Chapitre 16, dédié à la correction appliquée à la reconnaissance des caractères.

15.1 Postulats

Nos postulats se fondent sur les points faibles relevés d'une part dans l'état de l'art (cf. Chapitre 14), et d'autre part dans les modules linguistiques du synthétiseur tels qu'ils ont été présentés (cf. Chapitre 13). Ils concernent la typologie des erreurs, mais également l'architecture de l'analyse morpho-syntaxique.

15.1.1 Typologie des erreurs

Postulat 15.1.1 (Erreurs audibles)**.** *En synthèse, un système de correction doit se concentrer sur les erreurs d'orthographe qui sont décelables dans le flux de la parole, parce qu'elles en rendent l'écoute inconfortable. Ces erreurs sont celles qui corrompent la forme acoustique d'un mot ou qui influencent la courbe prosodique d'une phrase.*

15.1.2 Architecture de l'analyse

Postulat 15.1.2 (Approche globale)**.** *La correction orthographique doit faire partie intégrante du processus d'analyse linguistique, de manière à profiter au maximum de l'ensemble des informations disponibles. Le processus de correction peut donc proposer des solutions en cours de traitement, mais doit attendre la fin du traitement avant de prendre une décision.*

Postulat 15.1.3 (Gestion des treillis)**.** *Etant donné une erreur, les solutions trouvées par une étape de correction peuvent en proposer un découpage différent. Il devient dès lors difficile de les mémoriser dans la structure de données du synthétiseur, dans l'attente d'un choix ultérieur.*

Postulat 15.1.4 (Limitation des transferts de données)**.** *Tout traitement doit éviter de stocker dans la structure de données des informations qui seront modifiées ou supprimées par les traitements suivants. Respecter cette contrainte limite le temps imparti à la conversion des données entre traitement et structure de stockage.*

15.2 Des postulats à l'algorithme

15.2.1 Au niveau de la typologie des erreurs

Notre Postulat 15.1.1 pose la nécessité, en synthèse, de traiter les erreurs audibles. Nous choisissons donc d'ignorer certaines erreurs.

15.2.1.1 Erreurs non gérées

Nous estimons que le système de synthèse se doit de respecter le style et le registre choisis par l'auteur. En somme, nous demandons au système de ne pas corriger les écarts volontaires de l'auteur par rapport à la norme. Parmi ces écarts, on compte par exemple les mots tronqués, qui miment l'oralité (*démo*, *'blème*). Ce genre de phénomènes est fréquent dans les rédactions informelles, comme le courier électronique.

En outre, étant donné que notre système de synthèse n'inclut aucun module sémantique et que ce domaine de recherche n'est pas notre spécialité, nous avons exclu de la correction toute erreur qui n'est détectable qu'à l'aide d'informations sémantiques. Nous ne gérons donc pas les listes de confusions purement phonétiques (*septique* >< *sceptique*, etc.), puisque ces erreurs ne sont pas audibles.

15.2.1.2 Erreurs gérées

Notre objectif est de traiter :

1. Les erreurs typographiques dont le résultat est un OOV. Ces erreurs sont très souvent audibles (*élèbe*, *couvet*, *soucvent*). Elles incluent les erreurs de casse et d'accentuation, que nous traiterons séparément, à l'aide de procédés plus légers (Hypothèse 15.2.4).
2. Les erreurs d'usage dont le résultat est un OOV. Dans la suite de ce document, nous parlons d'*erreurs phonétiques*, parce qu'elles demandent une correction phonétique et que l'utilisateur, qui ne connaît pas l'orthographe d'un mot, la déduit de sa prononciation (*auvent* → *ovant*, *pleine* → *plène*). Ces erreurs gênent l'analyse syntaxique.
3. Les véritables OOVs, qui sont des emprunts, des néologismes, mais également des termes absents à tort de nos lexiques. L'objectif est de « récupérer au mieux » les termes absents, en les soumettant à une analyse morphologique de qualité. L'objectif est double : faciliter l'analyse syntaxique, et donner l'information la plus pertinente possible à la phonétisation.
4. Les noms propres et les acronymes, inconnus de nos lexiques et pris à tort pour des OOVs.
5. Les erreurs d'accord, que l'erreur soit produite sur un IV ou sur un OOV. Ces erreurs sont audibles si elles impliquent une modification des liaisons ou de la prononciation (*quelque amis*, *petit fille*).

15.2.2 Au niveau de l'architecture de l'analyse

Afin de mettre en œuvre les Postulats 15.1.3 et 15.1.4, nous faisons les hypothèses suivantes :

Hypothèse 15.2.1 (Gestion des treillis)**.** *Les machines à états finis permettent de conserver les solutions d'un traitement sous la forme d'un treillis où les divergences de découpage importent peu.*

Hypothèse 15.2.2 (Limitation des transferts de données)**.** *Les machines à états finis évitent les transferts inutiles entre le traitement et le stockage, en conservant l'ensemble des informations dans un format directement utilisable d'un traitement à l'autre.*

Nous basant sur ces hypothèses, nous avons réalisé une modification en profondeur de l'analyse linguistique d'eLite :

1. Nous avons fusionné les modules d'analyse morphologique et d'analyse syntaxique en un seul module d'analyse morpho-syntaxique. Ceci permet de proposer une approche globale de la correction orthographique (Postulat 15.1.2) et supprime la nécessité de sauvegardes intermédiaires dans la structure de données (Postulat 15.1.4). L'analyse linguistique n'isole dès lors plus que le pré-processeur dans un module séparé.

2. Le texte à analyser, ainsi que la *totalité* des modèles et des dictionnaires utilisés sont des machines à états finis. Ceci permet aux différents traitements de communiquer directement, sans recourir à la DLS (Postulat 15.1.4), facilite la gestion des treillis de solutions (Postulat 15.1.3) et homogénéise l'ensemble du module. Les modèles et les dictionnaires ont été compilés à l'aide de notre compilateur, Ovide, qui a été présenté en Section 6.2 et dont la documentation se trouve en Annexe B.

15.2.3 Au niveau du modèle de correction

Dans l'optique de simplifier la mise en œuvre de la correction dans l'analyse morpho-syntaxique, l'algorithme proposé repose également sur les hypothèses suivantes :

Hypothèse 15.2.3 (Approche stratifiée)**.** *La correction orthographique peut utilement s'organiser en couches, de manière à corriger progressivement les formes qui contiennent différents types d'erreurs.*

Hypothèse 15.2.4 (Casse et accentuation)**.** *Il est fréquent qu'une forme lexicale ne soit pas reconnue par l'analyse alors qu'elle ne diffère de la forme recensée dans le lexique qu'au niveau de la casse ou de l'accentuation. Il est dès lors important de détecter ces formes au plus tôt, afin de leur éviter un traitement lourd à réserver aux véritables erreurs typographiques.*

15.3 L'algorithme

Note 15.3.1. L'objectif de cette section est de donner un aperçu global et compréhensible de l'ensemble du processus. Les choix réalisés ne sont de ce fait pas justifiés dans cette section. Ces questions sont abordées lorsque nous décrivons les modèles qui font l'originalité de l'approche.

Note 15.3.2. Le processus complet de l'analyse morpho-syntaxique implique de nombreuses étapes qui, pour la clarté du code, ont été réparties en différentes sous-routines. L'algorithme est de ce fait relativement difficile à cerner dans son ensemble, si la présentation qui en est faite reste trop proche de l'implémentation. Pour cette raison, nous illustrons l'algorithme général et les grands traitements qu'il implique au travers de schémas-blocs. Les pseudocodes correspondants sont quant à eux placés en Annexe 4.

15.3.1 Algorithme complet

La Figure 15.1 présente le schéma-bloc de l'analyse morpho-syntaxique dans son ensemble. Une boucle gère la totalité du processus. A chaque itération, un ou plusieurs tokens entiers sont traités, de sorte qu'à chaque itération, on se situe toujours sur le premier mot du token courant. La première itération est réalisée sur le premier token de la DLS et la boucle s'interrompt lorsque le dernier token de la DLS a été traité.

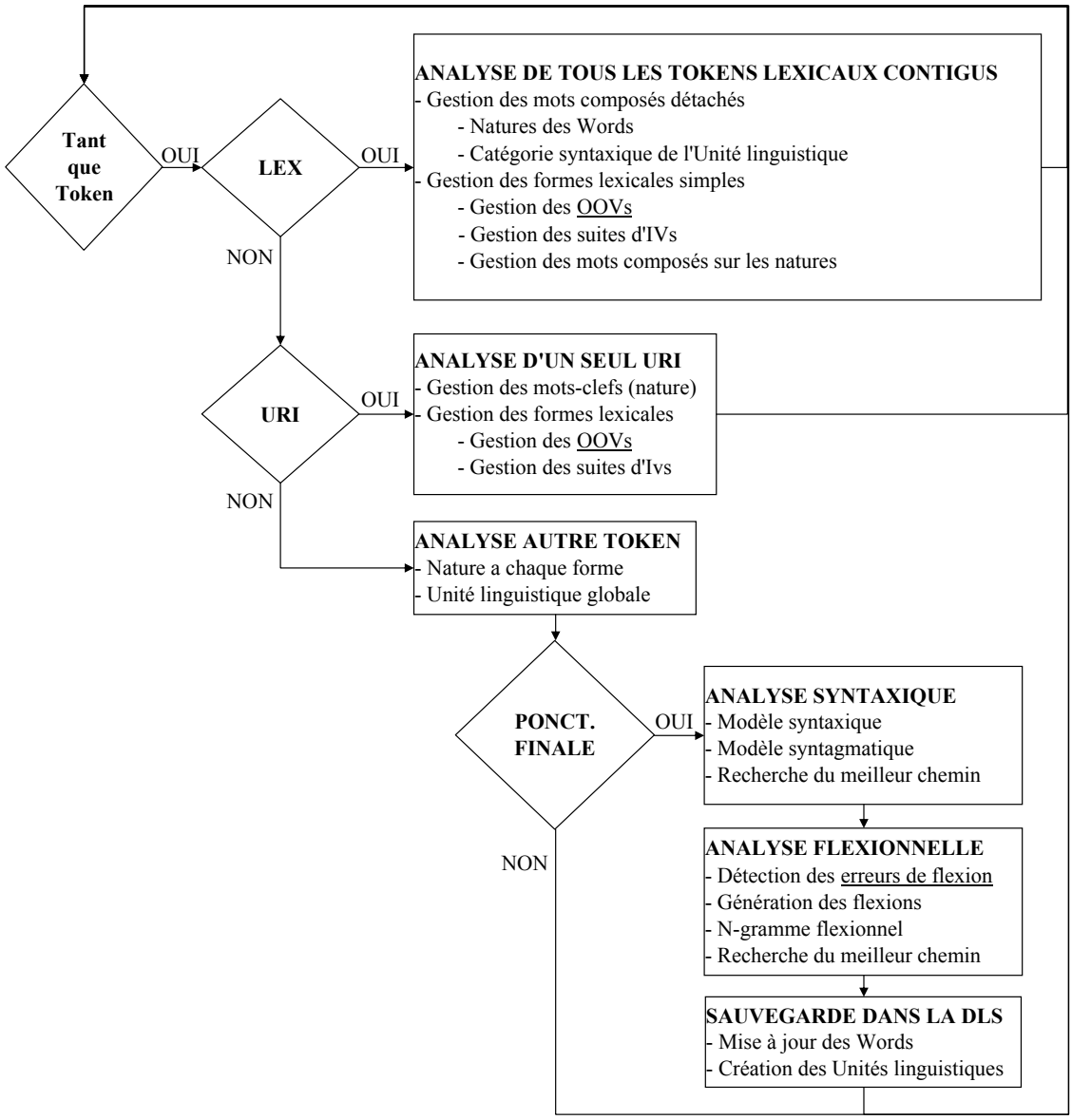


FIG. 15.1: Schéma-bloc de l'algorithme général

A chaque itération, on teste le type du token courant. Comme dans l'ancien analyseur morphologique, le traitement varie selon que l'on se trouve sur un token lexical, une URI ou au autre token. Ce principe respecte en effet totalement notre Hypothèse 1, qui décrit l'intérêt de spécialiser le traitement en fonction des données. Dans le cas d'un token lexical, le traitement concerne la *totalité des tokens lexicaux contigus* : ceci facilite la gestion des mots composés lexicaux. Les URIs et les autres tokens sont par contre traités un à un, étant donné qu'ils constituent des unités linguistiques autonomes. Quel que soit le type de

token traité, le traitement réalisé renvoie toujours le résultat de son analyse sous la forme d'un FSM.

Contrairement à l'ancien système, l'analyse syntaxique est incluse dans le processus. Elle est exécutée lorsque le token est une *ponctuation finale* [1]. L'analyse syntaxique de chaque phrase est donc réalisée séparément. Les différents FSMs retournés par les traitements précédents sont concaténés ensemble, de sorte que toutes les variantes possibles de la phrase à analyser sont représentées sous la forme d'un seul FSM. Comme précédemment, l'analyse syntaxique est réalisée à l'aide d'un modèle de langue. Cependant, contrairement à l'ancien système, le modèle de langue a été scindé entre l'analyse morphologique et l'analyse syntaxique. L'analyse syntaxique est maintenant limitée au modèle syntaxique, que nous avons complété par un nouveau modèle syntagmatique. L'analyse syntaxique retourne un FSM qui contient le chemin le plus probable dans le treillis de possibilités reçu.

La phase d'analyse syntaxique est directement suivie de l'analyse flexionnelle. Seul le meilleur chemin du treillis syntaxique est donc pris en compte pour cette phase de correction. La correction flexionnelle tâche de détecter les flexions fautives et de les corriger. Cette correction se base sur la définition d'un nouveau modèle de langue, qui inclut les flexions, mais limite le nombre de catégories utilisées.

Le traitement de la phrase se termine par une mise à jour de la DLS : un parcours du FSM retourné par l'analyse flexionnelle permet de mettre à jour les words qui le nécessitent, et de créer les unités linguistiques. L'itération suivante, s'il y en a une, entame une *nouvelle* phrase.

La correction orthographique principalement concerne les tokens lexicaux. Initialement, le traitement des tokens lexicaux s'occupait d'ailleurs de la détection des erreurs de flexion. Des tests réalisés en cours de modélisation du système nous ont cependant convaincu du *danger* de cette approche : en effet, l'insertion de variantes flexionnelles *avant* l'analyse syntaxique perturbe considérablement le système, dont les performances en terme d'étiquetage diminuent dans ce cas d'environ 3%. Ce constat est ainsi à la base de la gestion de la correction flexionnelle *après* l'analyse syntaxique.

Ci-dessous, nous décrivons, dans des points séparés, les algorithmes dédiés à l'analyse des tokens lexicaux, des URIs et des autres tokens.

15.3.2 Algorithme d'analyse des tokens lexicaux

La Figure 15.2 présente le schéma-bloc des traitements réalisés sur une suite de tokens lexicaux. La première étape consiste à construire un FSM avec l'ensemble des formes lexicales qui appartiennent à des tokens lexicaux contigus. Un test dichotomique permet d'y distinguer les formes lexicales simples de celles qui constituent des composés détachés. Sur cette base, le FSM est segmenté dans un vecteur de FSMs dont chaque position est soit *un*

[1] Nous rappelons que la présence d'une ponctuation finale à la fin d'une phrase est assurée par le préprocesseur, qui en insère une lorsqu'elle manque (cf. Section 13.3).

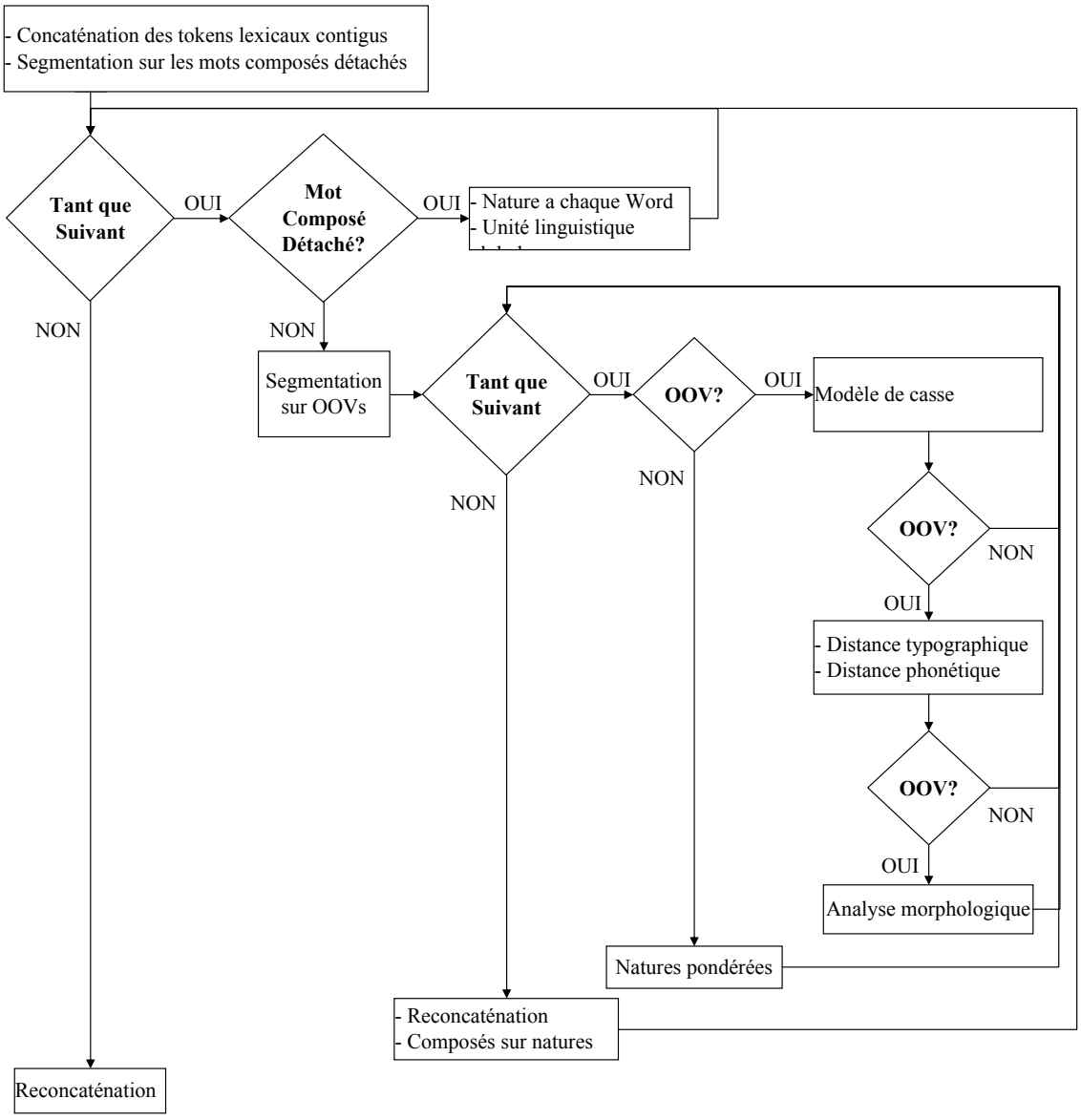


FIG. 15.2: Schéma-bloc de l'analyse des tokens lexicaux

seul mot composé détaché, soit *une suite* de formes lexicales simples. Les formes lexicales simples restent donc dans un FSM commun. Les différentes positions du vecteur sont traitées dans une boucle où l'analyse varie selon que le FSM courant est un mot composé ou une suite de formes lexicales.

A la sortie de la boucle, le vecteur est reconcaténé en un seul FSM, qui contient la totalité des analyses au niveau *word* et au niveau *unité linguistique*. Le résultat est retourné à l'algorithme principal.

Analyse d'un mot composé détaché. Un traitement léger est appliqué : chacun des mots du composé se voit attribuer une nature et une analyse flexionnelle, et l'ensemble du mot composé reçoit une catégorie syntaxique, qui correspond à son unité linguistique.

Analyse d'une suite de formes lexicales simples. Le traitement, qui est beaucoup plus complexe que le précédent, repose entièrement sur un test dichotomique qui identifie dans le FSM les IVs et les OOVs. Sur cette base, le FSM est segmenté dans un vecteur de FSMs dont chaque position est soit *un seul* OOV, soit *une suite* d'IVs. Les différentes positions du vecteur sont traitées dans une boucle où l'analyse varie selon que le FSM courant est un OOV ou une suite d'IVs.

Le traitement d'une suite d'IVs est fort simple. En une seule opération, chaque mot de la suite se voit attribuer l'ensemble des paires {nature grammaticale pondérée, analyse flexionnelle} qui lui correspondent et qui sont mémorisées dans un simple dictionnaire.

Le traitement d'un OOV est de loin plus complexe. Un premier test s'intéresse à la casse de l'OOV. Si l'OOV commence par une majuscule ou est complètement en majuscule, les catégories *nom propre* et/ou *acronyme* lui sont attribuées et le traitement est interrompu. Sinon, le système tente de proposer des corrections sur la base d'une distance typographique et d'une distance phonétique. Si des corrections sont trouvées, *aucun choix* n'est réalisé, mais le traitement s'arrête là. Chaque correction est accompagnée de paires {nature grammaticale pondérée, analyse flexionnelle}. Dans le cas contraire, l'OOV est soumis à une analyse morphologique afin de déterminer de manière fiable les paires {nature grammaticale pondérée, analyse(s) flexionnelle(s)}. Dans ce cas-ci, une même nature grammaticale peut correspondre à plusieurs analyses flexionnelles.

Lorsque le vecteur complet des formes lexicales simples a été traité, un seul FSM est reconstitué. A ce niveau, chaque forme lexicale possède au moins une nature grammaticale. C'est donc le bon endroit pour tâcher de détecter les suites de natures qui participent à la création de composés détachés inconnus de nos dictionnaires.

15.3.3 Analyse d'une URI

La Figure 15.3 présente le schéma-bloc des traitements réalisés sur une URI. La première étape consiste à construire un FSM avec l'ensemble des words qui appartiennent au token. Un test dichotomique permet d'y distinguer les mots-clefs, des formes lexicales.

Note 15.3.3. Les mots-clefs sont en fait les mots réservés et les symboles qui identifient une URI de manière univoque et en organisent la syntaxe. Parmi les mots réservés, on trouve les protocoles (`http`, `ftp`, `mailto`, etc.), le Web (`www`), les extensions (`be`, `fr`, `com`, etc.). Parmi les symboles, on trouve le point (`.`), les simple et double barres obliques (`/`, `//`), les deux-points (:), etc.

Sur cette base, le FSM est segmenté dans un vecteur de FSMs dont chaque position est soit *une suite* de mots-clefs, soit *une seule* forme lexicale. Les mots-clefs restent donc

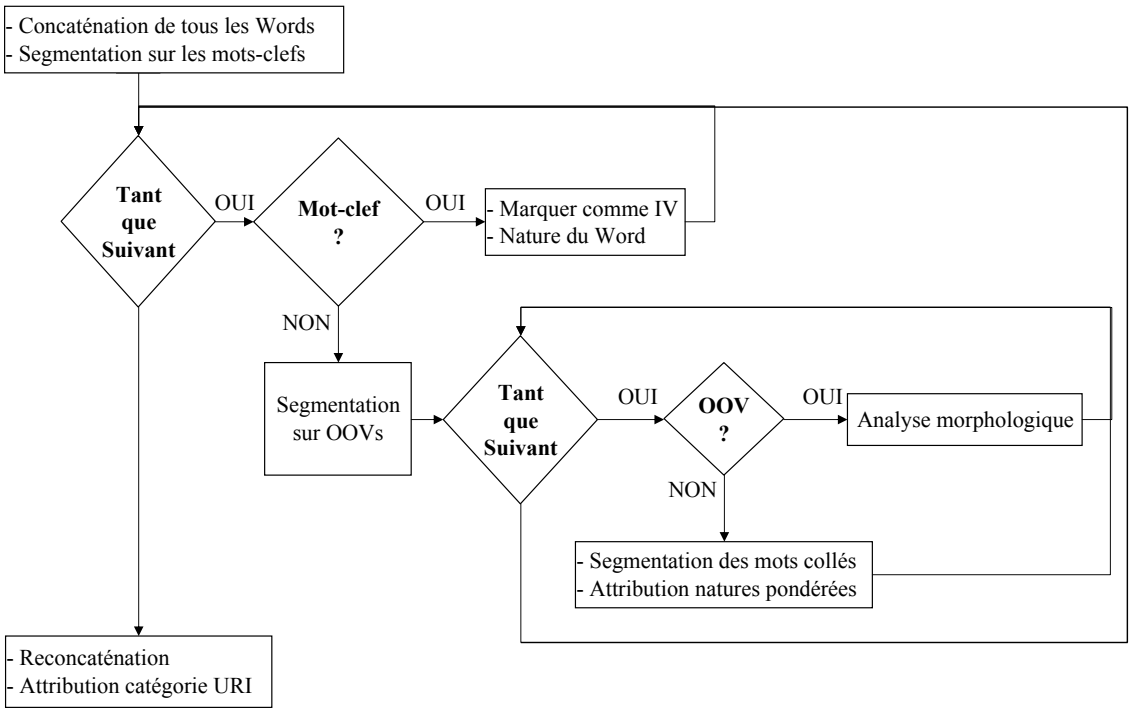


FIG. 15.3: Schéma-bloc de l'analyse des URIs

dans un FSM commun. Les différentes positions du vecteur sont traitées dans une boucle où l'analyse varie selon que le FSM courant est une forme lexicale ou une suite de mots-clefs.

A la sortie de la boucle, le vecteur est reconcaténé en un seul FSM, qui contient la totalité des analyses au niveau *word*. Le dernier traitement attribue l'unité linguistique `URI` à l'ensemble du FSM. Le résultat est retourné à l'algorithme principal.

Traitement d'une suite de mots-clefs. La seule opération est un accès à un dictionnaire spécialisé qui attribue, en une seule opération, la nature grammaticale qui correspond à chaque mot-clef. Par exemple, `http` sera analysé `ACRONYM`, tandis que // sera étiqueté `SYMBOL`.

Traitement d'une forme lexicale. L'analyse est un peu plus complexe, parce qu'il est fréquent qu'une forme lexicale, dans une URI, corresponde à plusieurs mots collés (`rendezvous` [2], `ilferabeaudemain` [3], `chaletneigesoleil` [4], etc.). Un test dichotomique détermine si la forme correspond à un ou plusieurs IVs collés, ou à un OOV.

Il faut noter les IVs collés sont automatiquement séparés par un espace au cours

[2] www.rendez-vous.be/.
[3] www.ilferabeaudemain.be.
[4] www.chaletneigesoleil.com/.

du test dichotomique. De ce fait, un seul traitement est appliqué à une forme IV, qu'il s'agisse d'un seul ou de plusieurs mots collés : en une seule opération, une recherche dans le dictionnaire attribue à chaque mot les paires {nature grammaticale pondérée, analyse flexionnelle} qui lui correspondent. On notera que, contrairement à la première version de l'analyse morphologique, la nouvelle analyse possède maintenant un outil qui lui permet de gérer les formes lexicales collées...

Dans le cas d'un OOV, le système réalise la même analyse morphologique que celle réalisée sur les OOVs des tokens lexicaux. La différence est que, dans le cas d'une URI, aucune correction n'est tentée avant l'analyse morphologique : nous partons du principe que la correction serait trop aléatoire, dans une unité où les noms propres et les mots étrangers sont fort fréquents.

15.3.4 Analyse des autres tokens

Le traitement appliqué aux autres tokens est fort simple. Etant donné que le traitement concerne tous les autres types de tokens, le processus commence par déterminer le dictionnaire spécialisé à utiliser. Ce dictionnaire est un FSM.

Ceci étant fait, le système construit un FSM avec l'ensemble des words qui appartiennent au token. Ensuite, l'accès au dictionnaire choisi permet d'attribuer, en une seule opération, la nature grammaticale qui correspond à chaque word du token.

Enfin, l'unité linguistique adéquate est attribuée au FSM, et celui-ci est retourné à l'algorithme principal. Le processus d'attribution de l'unité linguistique à un token est décrit en Section 15.4.

15.4 Lignes de faîte de l'algorithme

Les traitements décrits dans la section précédente sont réalisés exclusivement à l'aide de machines à états finis. Ils reposent sur la définition d'une interface de communication commune, et sur le développement de méthodes autorisant de nouvelles manipulations des machines à états finis.

15.4.1 Interface de communication

Afin de mettre en œuvre notre Postulat 15.1.4, qui met en avant l'importance de limiter les transferts entre les traitements et la structure de données, nos traitements ne sauvent aucune information dans la DLS tant que le traitement complet d'une phrase n'est pas terminé. Ceci implique que :

1. Les FSMs doivent être définis sur un *alphabet commun*, afin que les entiers qui étiquettent les transitions des différentes machines conservent la même valeur *symbolique* d'un traitement à l'autre.

2. Les traitements reçoivent et retournent des FSMs qui respectent toujours un *format défini*.

15.4.1.1 L'alphabet

L'alphabet contient la *totalité* des symboles nécessaires aux divers traitements. Ceci implique :

1. Les caractères ASCII [5], étant donné que l'on traite du texte.
2. Les natures grammaticales et catégories syntaxiques, dont la liste pour le français est détaillée en Annexe 1. Il faut noter que les natures et les catégories de même valeur partagent le même symbole. Par exemple, `NOUN` et `VERB` peuvent autant être natures grammaticales que catégories syntaxiques. Il faut cependant pouvoir différencier les natures et les catégories dans les machines. Nous détaillons ceci dans le point suivant.
3. Les classes flexionnelles et les traits grammaticaux des formes lexicales, dont la liste se trouve en Annexe 3.
4. Les symboles qui identifient certaines caractéristiques des formes, comme `IV`, `OOV`...

En tout, l'alphabet de l'analyseur morpho-syntaxique compte environ 560 symboles.

15.4.1.2 Le format des machines

Le format défini repose sur les constats suivants :

1. Les couches *Word* et *Unité linguistique* de la DLS sont traitées par l'analyse morpho-syntaxique. Or, les natures grammaticales et les catégories syntaxiques partagent des symboles communs. Il faut donc les distinguer par leur position dans la machine.
2. Il faut un séparateur sans ambiguïté entre les éléments de la couche *Word*.

Nous avons dès lors pris les décisions suivantes :

1. Nous réservons l'entrée de la machine aux informations de la couche *Word*, tandis que la sortie correspond à la couche *Unité linguistique*. L'entrée de la machine contient donc la forme lexicale, la nature grammaticale, la classe flexionnelle et les traits grammaticaux, tandis que la sortie de la machine contient exclusivement les catégories syntaxiques des unités linguistiques.
2. Les éléments de la couche Word sont séparés par un symbole qui ne peut apparaître nulle part ailleurs : l'espace.

Les différents traitements sont prévus pour construire ce format d'étape en étape, comme l'illustre la Figure 15.4 :

1. Un premier traitement construit toujours un transducteur identitaire, ne contenant que la ou les formes lexicales, séparées par des espaces.

[5] ASCII : *American Standard Code for Information Interchange*.

2. Un second traitement obtient,

   (a) D'abord un transducteur dont la sortie est augmentée de l'analyse grammaticale de chaque forme lexicale. Dans l'ordre : la classe flexionnelle, la désinence, la nature grammaticale. Classe et désinence ne sont pas ajoutées si le terme est invariable. L'opération principale de cette étape est la composition avec un modèle donné, mais de nombreuses autres opérations peuvent intervenir dans le processus.

   (b) Ensuite, un transducteur identitaire contenant les formes lexicales et l'analyse grammaticale. Une simple projection de la sortie du transducteur précédent permet d'obtenir ce résultat.

3. Un troisième traitement obtient un transducteur dont la sortie ne contient plus que des catégories syntaxiques. A ce niveau, l'entrée est donc dédiée aux words, et la sortie, aux unités linguistiques. Dans le transducteur, une catégorie syntaxique est toujours alignée sur la première nature grammaticale d'une unité.

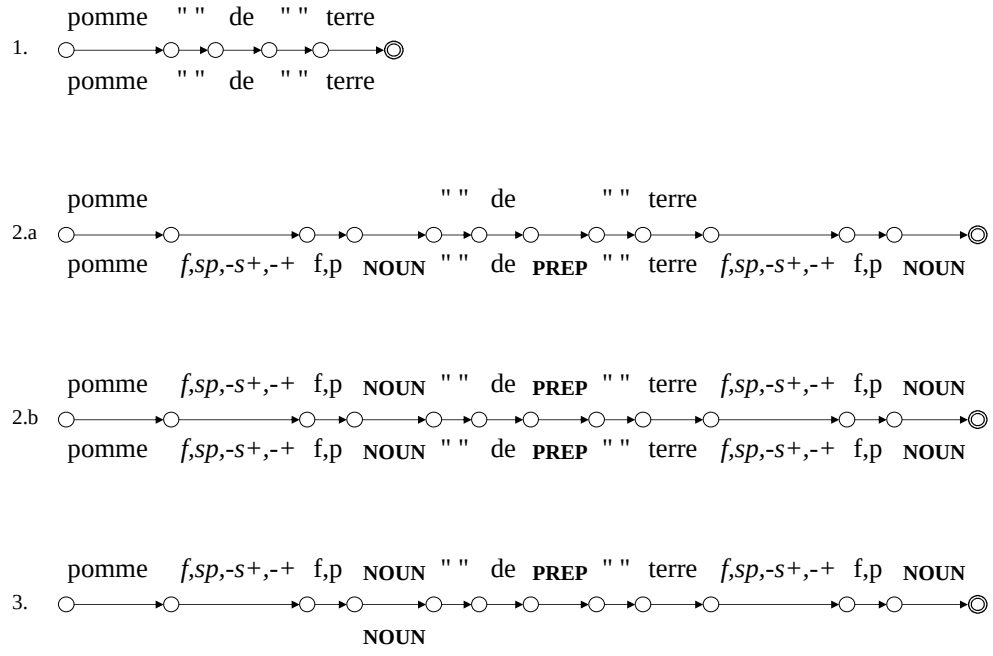


FIG. 15.4: Format des machines. Par manque de place et pour une plus grande clarté, les machines ont été simplifiées : les formes lexicales et les désinences sont représentées chacune sur une seule transition, alors que chaque caractère et chaque trait grammatical correspond en réalité à une transition. Les transitions ϵ, quant à elles, ne sont pas indiquées

15.4.2 Les méthodes

Trois méthodes ont été définies pour la bonne marche de l'algorithme dans son ensemble :

1. *La composition filtrée.* Il s'agit d'une méthode particulière de composition, mise en œuvre par la plupart des traitements réalisés.
2. *Le test dichotomique.* Son objectif est d'identifier deux types de données dans un FSM.
3. *La segmentation.* Un FSM qui contient des données dichotomiques est segmenté sur ce critère, de manière à adapter le traitement en fonction du type.

Notons que la notion de *composition filtrée* que nous introduisons repose sur la définition préalable de la *distance d'édition de deux automates.*

15.4.2.1 Distance d'édition de deux automates

La distance d'édition de deux automates a été proposée par Mohri (2003), et modélise, sous la forme de machines à états finis, la distance de Levenshtein (cf. Section 14.4.1). Cette modélisation n'a pas été présentée dans l'état de l'art de la correction orthographique, parce qu'elle n'a jamais été appliquée dans ce domaine. Mohri, quant à lui, mentionne l'applicabilité de la méthode en reconnaissance de la parole et en biologie informatique. Nous ne rappelons ici que les conclusions de la démonstration et laissons le soin au lecteur de consulter l'article concerné pour de plus amples explications.

Posons un alphabet Σ, et $\Omega = \Sigma \cup \{\epsilon\} \times \Sigma \cup \{\epsilon\} - \{\epsilon, \epsilon\}$, qui représente les opérations d'édition possibles à partir de Σ. Pour tout $a, b \in \Sigma$,

- (a, b) note la substitution,
- (ϵ, a) note l'insertion,
- (a, ϵ) note la suppression.

On note h l'homomorphisme entre un élément w de Ω^* et un couple de strings (x, y) de $\Sigma^* \times \Sigma^*$.

$$\begin{array}{l} \forall w = (a_1, b_1) \dots (a_n, b_n) \in \Omega^*, \forall (x, y) = (a_1 \dots a_n, b_1 \dots b_n) \in \Sigma^* \times \Sigma^*, \\ \quad h((a_1, b_1) \dots (a_n, b_n)) = (a_1 \dots a_n, b_1 \dots b_n) \\ \quad h(w) = (x, y) \end{array} \tag{15.4.2.1}$$

Cet homomorphisme permet de définir l'alignement de deux strings :

Définition 15.4.1 (Alignement)**.** *Un* alignement w *de deux strings* x *et* y *sur l'alphabet* Σ *est un élément de* Ω^* *tel que* $h(w) = (x, y)$.

Posons la fonction $c : \Omega \mapsto \mathbb{R}^+$ qui associe à chaque élément de Ω un poids non négatif. Sur cette base, on définit le coût de $w \in \Omega^*$ comme la somme des coûts de ses parties :

$$c(w) = \sum_{i=0}^{n} c(w_i) \tag{15.4.2.2}$$

La distance d'édition entre deux strings se définit dès lors comme suit :

Définition 15.4.2 (Distance d'édition)**.** *La* distance d'édition $d(x, y)$ *entre deux strings* x *et* y *sur l'alphabet* Σ *est le coût minimal d'une séquence d'insertions, de suppressions ou de substitutions transformant une string en une autre :*

$$d(x, y) = \min\{c(w) : h(w) = (x, y)\} \tag{15.4.2.3}$$

La définition de la distance d'édition peut être étendue de manière à mesurer la similarité de deux ensembles de strings X et Y par :

$$d(X, Y) = \inf\{d(x, y) : x \in X,\ y \in Y\} \tag{15.4.2.4}$$

Posons A_1 et A_2, les deux automates qui représentent les ensembles de strings X et Y. La distance d'édition de ces deux automates se définit comme suit :

$$d(A_1, A_2) = \inf\{d(x, y) : x \in Dom(A_1),\ y \in Dom(A_2)\} \tag{15.4.2.5}$$

Posons Ψ la série entière rationnelle qui projette Ω sur le semi-anneau tropical : $(\Psi, (a, b)) = c(a, b)$ pour tout $(a, b) \in \Omega$. Ψ^* correspond à la clôture de Kleene de Ψ. Le théorème de Schützenberger (cf. Section 4.2.2) assure l'existence d'un automate A qui réalise Ψ^*. A peut également être vu comme un transducteur pondéré T dont l'entrée et la sortie sont définis sur Σ. Ce transducteur représente les opérations d'édition entre deux automates définis sur Σ. La distance d'édition entre A_1 et A_2 peut dès lors être calculée à l'aide de l'algorithme de composition défini sur les transducteurs (cf. Section 3.3.3) et de l'algorithme de recherche du meilleur chemin (cf. Section 4.5). La composition entre A_1 et A_2 correspond dès lors à :

$$U = A_1 \circ T \circ A_2 \tag{15.4.2.6}$$

Par définition, U contient un chemin valide correspondant à chaque alignement w entre une string acceptée par A_1 et une string acceptée par A_2. La distance d'édition $d(A_1, A_2)$ correspond au meilleur chemin π de U, dont le poids vaut $c(\pi)$.

Complexité. Mohri note que lorsque A_1 et A_2 sont tous les deux acycliques, la complexité de $d(A_1, A_2)$ vaut $O(|A_1||A_2|)$.

Coût d'édition. Dans la définition classique de la distance d'édition, le coût de toute opération d'édition vaut 1 :

$$\forall a, b \in \Sigma,\ c((a,b)) = 1 \text{ si } a \neq b,\ 0 \text{ sinon} \tag{15.4.2.7}$$

Le transducteur T_{ed1}, qui réalise la distance d'édition classique où toute opération vaut 1, est illustré en Figure 15.5 pour $\Sigma = \{a, b\}$. Mohri note cependant que, sans modifier

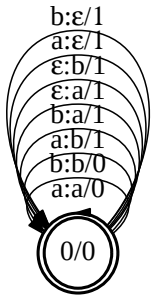


FIG. 15.5: Transducteur T_{ed1}. Figure reprise de (Mohri 2003)

l'algorithme, tout transducteur pondéré par des poids non négatifs peut être utilisé à la place de T_1 afin de modéliser des distances d'édition plus complexes.

15.4.2.2 Composition filtrée

La composition filtrée peut être considérée comme une généralisation de la notion de distance d'édition de deux automates. Cette généralisation concerne :

1. L'extension de la distance d'édition aux transducteurs.
2. La définition de filtres de composition en cascade.

Distance d'édition de deux transducteurs. L'extension de la distance d'édition aux transducteurs se base sur les propriétés de la composition ainsi que sur la notion de filtre de composition proposée par Pereira & Riley (1997) (cf. Section 3.3.3).

Posons deux transducteurs lettre-à-lettre $T_1 = (\Sigma_1, \Sigma, Q_1, i_1, F_1, E_1)$ et $T_2 = (\Sigma, \Sigma_2, Q_2, i_2, F_2, E_2)$. T_1 représente la transduction entre les ensembles de strings X et Z_1, T_2 représente la transduction entre les ensembles de strings Z_2 et Y. La distance d'édition entre ces deux transducteurs se définit comme suit :

$$d(T_1, T_2) = \inf\{d(x, y) : (x{:}z_1) \in Dom(T_1),\ (z_2{:}y) \in Dom(T_2)\} \tag{15.4.2.8}$$

Le transducteur T, qui réalise Ψ^* et est défini sur l'alphabet Σ, peut être considéré comme un filtre de composition pondéré. La distance d'édition entre T_1 et T_2 correspond dès lors au meilleur chemin de leur composition au travers de ce filtre pondéré :

$$V = T_1 \circ T \circ T_2 \tag{15.4.2.9}$$

Par définition, V contient un chemin valide (x, y) correspondant à chaque alignement w entre une string z_1 acceptée par la seconde projection de T_1, et une string z_2 acceptée par la première projection de T_2. La distance d'édition $d(T_1, T_2)$ correspond au meilleur chemin π de V, dont le poids vaut $c(\pi)$.

Filtres de composition en cascade. Mohri note que tout transducteur pondéré peut être utilisé pour calculer la distance d'édition de deux automates, sans modification de l'algorithme. Ce transducteur pondéré T peut dès lors être lui-même le résultat d'une cascade de transductions. Par exemple,

$$T = T_3 \circ T_4$$

Ceci étant posé, l'associativité de la transduction posée en Section 3.3.3 assure que :

$$\begin{aligned} T_1 \circ T \circ T_2 &= T_1 \circ (T_3 \circ T_4) \circ T_2 \\ &= (T_1 \circ T_3) \circ (T_4 \circ T_2) \\ &= ((T_1 \circ T_3) \circ T_4) \circ T_2 \end{aligned}$$

La loi d'associativité de la composition permet donc de représenter un filtre de composition sous la forme d'une cascade de transductions. Deux avantages s'en déduisent directement :

1. *Flexibilité du filtre.* Les différentes machines qui interviennent dans la construction du filtre restent indépendantes. Le filtre complet ne doit donc pas être recalculé à chaque modification de l'une de ses parties.
2. *Flexibilité de l'algorithme.* Le calcul de la distance d'édition entre deux machines à états finis peut être réparti entre plusieurs points d'un même algorithme.

Objectif de la composition filtrée. Dans notre modèle de correction, la composition filtrée n'a pas pour unique vocation de déterminer la distance d'édition entre deux machines à états finis. De manière plus générale, son objectif est de *modifier* la taille de l'intersection des langages sur lesquels est calculée une composition. Le filtre de composition augmente ou diminue les possibilités de composition entre deux machines à états finis.

Définition 15.4.3 (Filtre de composition permissif)**.** *Un filtre de composition est dit* permissif *s'il augmente la taille de l'intersection des langages auxquels il s'applique.*

Définition 15.4.4 (Filtre de composition restrictif)**.** *Un filtre de composition est dit* restrictif *s'il diminue la taille de l'intersection des langages auxquels il s'applique.*

Définition 15.4.5 (Composition filtrée, permissive et restrictive)**.** *Une composition filtrée entre deux machines à états finis M_1 et M_2 est une composition qui implique une composition intermédiaire avec un filtre de composition. La composition est dite* permissive *si le filtre est permissif, et* restrictive *si le filtre est restrictif.*

Composition relâchée. L'analyse morpho-syntaxique recourt fréquemment à un cas particulier de composition permissive, que nous appelons *composition relâchée*, afin de l'identifier de manière non ambiguë. Cette composition relâchée modélise, sous la forme d'une machine à états finis, la recherche relâchée qui a été décrite en Section 13.4.1.2.

La recherche relâchée permettait d'ignorer les différences de casse et d'accentuation lors d'un parcours du trie de flexions. De même, la composition relâchée permet d'accepter, dans l'intersection de la composition, des différences de casse et d'accentuation.

Un extrait des règles de ce filtre permissif est présenté en Figure 15.6. Des classes de symboles sont définies en entête du fichier et sont utilisées dans certaines règles entre *square brackets*. Les règles sont facultatives et pondérées (cf. Section 5.3).

```
[CLASSIN]
  A2      [àâäãáa]
  A3      [ÁÀÄÃÂ]
  A       [<A2>A]
  ...
[RULE]
  A ?→ <A> /1
  Á ?→ á / 1
  À ?→ à / 1
  Ä ?→ ä / 1
  Ã ?→ ã /1
  Â ?→ â /1
  <A3> ?→ <A> /2
  <A2> ?→ <A2> /2
  ...
```

FIG. 15.6: Fichier Ovide : recherche relâchée

Après composition avec ces règles, le FSM de la forme lexicale sera augmenté de nouvelles transitions pondérées, mais la forme originale sera toujours présente dans la machine, du fait de l'optionalité des règles. Après composition avec le FSM désiré, les différentes solutions trouvées pourront être classées grâce à la pondération des règles.

On notera que si la forme originale fait partie du résultat, elle en constitue la meilleure solution. En outre, il est évidemment possible de limiter le résultat aux n meilleures solutions selon les besoins du traitement en cours. Ceci est une différence fondamentale avec la recherche relâchée, qui ne permettait pas de classer les solutions.

La composition relâchée a été développée pour mettre en œuvre notre Hypothèse 15.2.4, qui pose l'intérêt d'éviter de soumettre à un traitement lourd les formes qui ne diffèrent de la forme recensée dans le lexique qu'au niveau de la casse ou de l'accentuation.

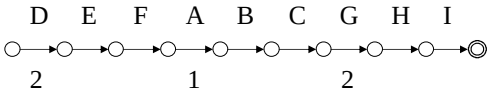


FIG. 15.7: Résultat d'un test dichotomique par alignement. ABC est de type 1, tandis que DEF et GHI sont de type 2

Note 15.4.1. Dans les équations ou pseudocodes qui ont recours à la composition relâchée, le filtre permissif qui gère les différences de casse et d'accentuation est toujours nommé $Match$.

15.4.2.3 Test dichotomique

Le test dichotomique permet d'identifier la présence de deux types de données dans un FSM. Ce test est réalisé par composition entre un FSM I, l'entrée à traiter, et un FSM D, qui modélise le test dichotomique :

$$I_D = I \circ D \tag{15.4.2.10}$$

Le modèle D projette une séquence de symboles sur le type auquel elle correspond. Le type est bien sûr également un symbole de l'alphabet. Les règles ont la forme générale suivante :

$$\begin{aligned} &R_1 \rightarrow t_1 \\ &R_2 \rightarrow t_2 \end{aligned} \tag{15.4.2.11}$$

où R_1 et R_2 sont des expressions régulières, et t_1 et t_2 sont les symboles des types correspondants. Le résultat de l'application de ces règles est un FST où le premier symbole d'une donnée est aligné sur le symbole du type correspondant, tandis que les autres symboles de la donnée sont alignés sur ϵ, comme l'illustre la Figure 15.7. Nous parlons de *dichotomie par alignement*.

La construction des règles est parfois nettement facilitée si l'on se contente d'identifier un seul des deux types. Cependant, dans ce cas, l'alignement n'est plus suffisant ; il faut *isoler* le type identifié, afin que les données de l'autre type puissent se déduire de l'analyse du résultat. Trois types de règles sont dès lors nécessaires : la première indique le début d'une donnée appartenant au type, la seconde indique la fin de cette donnée, et la troisième supprime de la sortie tout symbole qui n'est pas un indicateur du type :

$$\begin{aligned} &\backslash\texttt{x00} \ \rightarrow d \ :: \ \_ \ R \\ &\backslash\texttt{x00} \ \rightarrow f \ :: \ d \ R \ \_ \\ &[\hat{}d \ f] \rightarrow \backslash\texttt{x00} \end{aligned} \tag{15.4.2.12}$$

où R est le seul type de données identifié, d en identifie le début et f, la fin. Le résultat de l'application de ces règles est un FST où une donnée du type identifié est séparée des autres données par les symboles de début et de fin, comme l'illustre la Figure 15.8. Nous parlons de *dichotomie par isolement*.

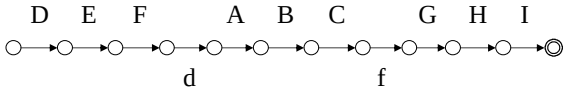


FIG. 15.8: Test dichotomique par isolement. ABC est de type 1. Par déduction, DEF et GHI sont de type 2

15.4.2.4 Segmentation

La segmentation divise un FSM I_D, qui correspond au résultat d'un test dichotomique, en un vecteur de FSMs, à l'aide d'une méthode que nous avons expressément ajoutée à la bibliothèque de machines à états finis,

$$I_{p,q} \texttt{ = FSM\_CopyPart(}I_D, p, q\texttt{)}$$

qui copie dans un FSM $I_{p,q}$ la partie de I_D qui commence à l'état p et termine à l'état q, et rend q final s'il ne l'était pas.

L'algorithme de segmentation varie selon qu'il s'agit d'une dichotomie par alignement ou par isolement, mais le principe reste fondamentalement le même. A titre d'exemple, le Pseudocode 56, situé en Annexe 5, présente l'algorithme de segmentation appliqué dans le cas d'une dichotomie par alignement.

Le principe de la segmentation n'est pas d'isoler *chaque forme* dans un FSM séparé. La méthode reçoit en argument une variable, qui lui précise le type pour lequel les séquences de formes doivent être préservées. Lorsque l'algorithme rencontre plusieurs formes contiguës du type en question, il évite de les séparer. C'est ainsi, par exemple, que l'algorithme morpho-syntaxique peut préserver les suites d'IVs lors la segmentation entre OOVs et IVs.

15.4.2.5 Application

Ensemble, le test dichotomique et la segmentation permettent de spécialiser le traitement réalisé en fonction du type identifié. Ces méthodes servent donc la mise en œuvre de notre Hypothèse 1, qui pose l'intérêt de soumettre des données de types différents à des traitements spécifiques.

La description de l'algorithme fait explicitement référence au test dichotomique et à la segmentation pour traiter séparément :

1. Les mots composés détachés et les formes lexicales simples.
2. Les IVs et les OOVs dans une séquence de formes lexicales simples.
3. Les mots-clefs et les formes lexicales dans une URI.
4. Les IVs (ou IVs collés) et les OOVs dans une forme lexicale d'une URI.

La méthode ne se limite cependant pas au test dichotomique et à la segmentation :

1. Notre Hypothèse 15.2.4 pose l'intérêt de détecter *au plus tôt* les erreurs d'orthographe qui se limitent à une simple différence de casse ou d'accentuation. Le test dichotomique entre une entrée I et le modèle D est de ce fait réalisé par composition relâchée :

$$I_D = I \circ Match \circ D \tag{15.4.2.13}$$

de manière à distinguer immédiatement les types qui ne seraient pas reconnus sans relâcher les contraintes de casse et d'accentuation. Ceci permettra, par exemple, de directement classer parmi les IVs des formes comme *Elève* (*élève*), *ELEVE* (*élève*|*élevé*) ou *aigue* (*aiguë*). Bien sûr, les formes qui comportent simultanément d'autres types d'erreurs seront confiées aux soins d'une correction plus lourde.

2. La segmentation doit être temporaire : le traitement, une fois terminé, doit retourner un FSM similaire à celui reçu en entrée. Une étape de reconcaténation des FSMs du vecteur est donc nécessaire.

La procédure complète est donc la suivante (cf. Figure 15.9) :

1. Test dichotomique par composition relâchée entre l'entrée et le modèle.
2. Segmentation du FSM selon le résultat du test.
3. Traitement des différentes parties du vecteur.
4. Reconcaténation des parties du vecteur en un seul FSM.

Dichotomie par isolement. Les mots composés sont les seuls à être identifiés à l'aide de la dichotomie par isolement. Les autres tests dichotomiques sont réalisés par alignement.

Pour les mots composés, l'isolement était plus facile à mettre en œuvre, parce que le dictionnaire de mots composés compte à peine 690 cas, et que l'on peut considérer que « tout ce qui n'est pas un mot composé est une suite de formes lexicales simples ». La Figure 15.10 illustre le résultat de la dichotomie par isolement, et de la segmentation qui en résulte, sur la séquence de formes lexicales « les pommes de terre sont cuites ». On constate sur la figure que l'algorithme de segmentation a reçu la consigne de préserver les suites de lexèmes.

Dichotomie par alignement. La dichotomie par alignement est idéale lorsque le modèle ne s'applique pas à Σ^*, mais à un sous-langage $M \subset \Sigma^*$.

Le test dichotomique qui, sur les séquences des formes lexicales simples, distingue les IVs des OOVs, en est un bon exemple. Il se fonde sur le fait que l'espace est l'élément séparateur des formes à étiqueter. Un espace ne fait donc *jamais* partie d'un IV ni d'un OOV. Le modèle se construit de ce fait comme suit :

1. Compilation d'un FSA L qui représente le dictionnaire des IVs, généré à partir de nos lexiques de lemmes et de classes flexionnelles (cf. Section 13.4.1.2).

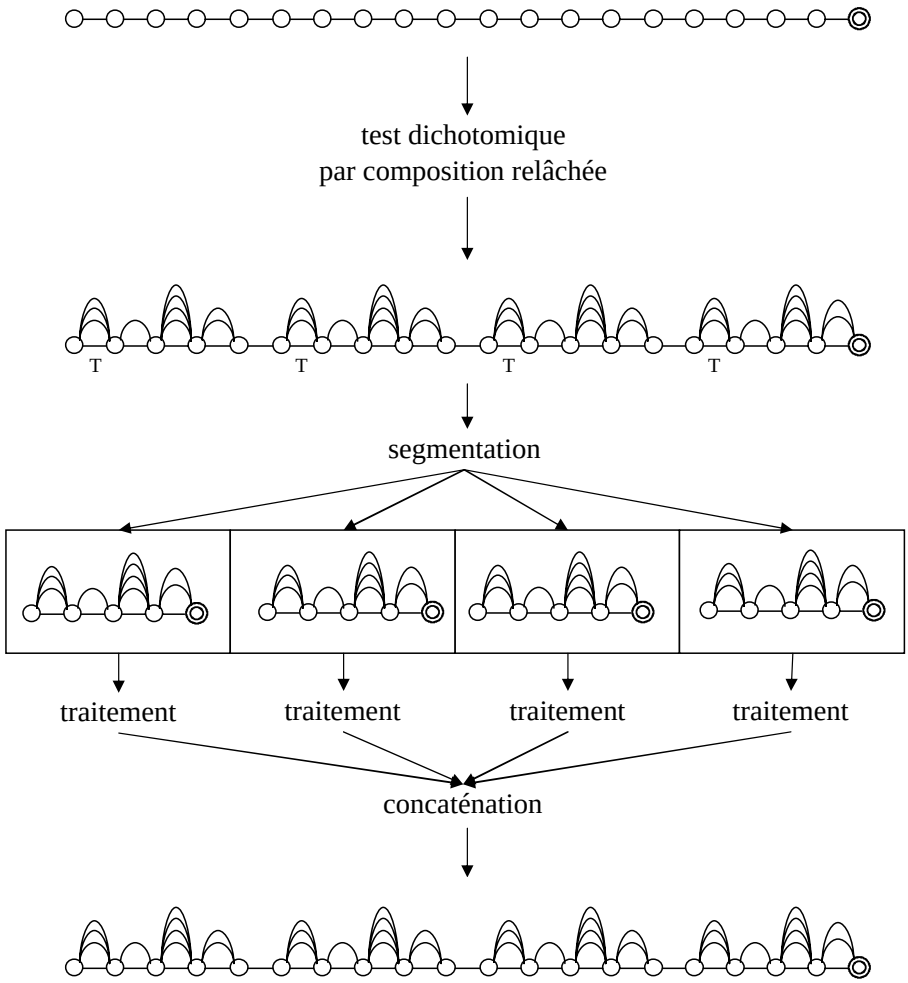


FIG. 15.9: Processus général de traitement des types de données différents : test dichotomique, segmentation, traitements et reconcaténation

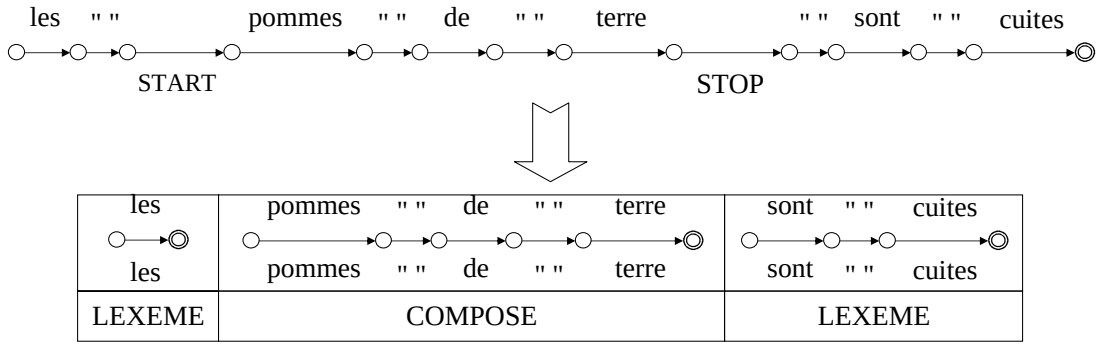


FIG. 15.10: Mots composés détachés : dichotomie par isolement

2. Compilation de $\overline{L}$, le complément de L, qui correspond à toutes les suites de caractères ASCII qui n'apparaissent pas dans L.
3. Etant donné que $\overline{L}$ contient des séquences comportant l'espace, il est nécessaire de le composer avec un FSM qui les supprime :

$$\overline{LS} = \overline{L} \circ S \tag{15.4.2.14}$$

où S correspond au langage $[\hat{} \setminus s]^*$, « toute suite de caractères sans espace ».

4. Compilation de $L_{\overrightarrow{IV}}$, qui projette toute forme IV sur le symbole `IV`. $L_{\overrightarrow{IV}}$ est le résultat de la composition suivante :

$$\mathrm{L}_{\overrightarrow{IV}} = L \circ \overrightarrow{IV} \tag{15.4.2.15}$$

où $\overrightarrow{IV}$ correspond à la règle de réécriture $.^* \rightarrow$ `IV`, qui projette toute suite de symboles sur le symbole `IV`.

5. Compilation de $\overline{L}_{\overrightarrow{OOV}}$, qui projette toute forme OOV sur le symbole `OOV`. $\overline{L}_{\overrightarrow{OOV}}$ est le résultat de la composition suivante :

$$\overline{LS}_{\overrightarrow{OOV}} = \overline{LS} \circ \overrightarrow{OOV} \tag{15.4.2.16}$$

où $\overrightarrow{OOV}$ correspond à la règle de réécriture $.^* \rightarrow$ `OOV`, qui projette toute suite de symboles sur le symbole `OOV`.

6. Compilation du modèle M, qui correspond à l'expression régulière suivante :

$$M = (L_{\overrightarrow{IV}} | \overline{LS}_{\overrightarrow{OOV}})\,(\backslash s\,(L_{\overrightarrow{IV}} | \overline{LS}_{\overrightarrow{OOV}}))^*$$

qui accepte en entrée une ou plusieurs formes lexicales séparées par des espaces. Chaque forme peut être IV ou OOV et est projetée, en sortie, sur le symbole qui lui correspond.

La Figure 15.11 illustre le résultat de la dichotomie par alignement, et de la segmentation qui en résulte, sur la séquence de formes lexicales « les élèbves sopnt bien élevés ».

On constate sur la figure que l'algorithme de segmentation a reçu la consigne de préserver les suites d'IVs.

La gestion du complément du dictionnaire a un impact énorme sur la taille de ce test dichotomique, qui correspond initialement à un FSM d'un peu plus de 98 Mo. Cependant, la compaction de cette machine à l'aide de classes de symboles permet de réduire sa taille à 5,5 Mo. Les classes de symboles qui peuvent être employées dans nos FSMs ont été décrites en Section 6.1.8.1. Dans le principe, nous rappelons qu'une classe de symboles regroupe, en une seule transition, l'ensemble des symboles qui étiquettent les transitions de même poids reliant deux états d'une machine.

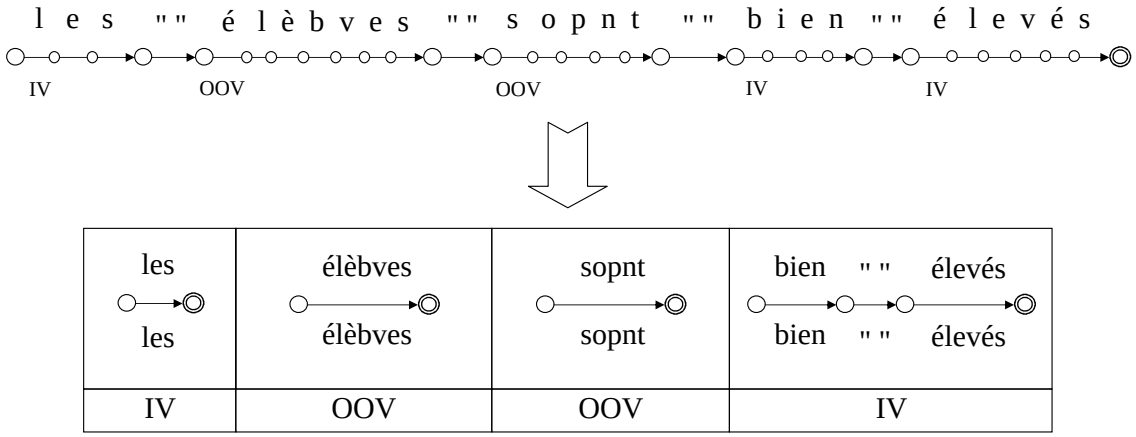


FIG. 15.11: distinction des IVs et des OOVs. L'alignement entre les formes et leurs types facilite la segmentation de la machine dans un vecteur de FSMs

15.5 Les modèles de l'analyse morpho-syntaxique

La mise en œuvre de l'ensemble de l'analyse morpho-syntaxique a été régie par l'impératif de parvenir à une fusion des deux analyses, sans *involontairement* modifier leur comportement initial. Dans cette entreprise, le plus délicat était de convertir notre modèle de langue sous la forme de machines à états finis.

Cette section s'ouvre par un survol des raisons qui ont conduit à une scission du modèle de langue entre l'analyse morphologique et l'analyse syntaxique. Elle se poursuit par une description du modèle de casse permettant la gestion des noms propres et des acronymes, avant de détailler les modèles dédiés à l'analyse morphologique. La section aborde ensuite la génération des unités linguistiques, et se referme sur une synthèse rapide des points abordés.

15.5.1 Modèle de langue

Le modèle de langue que nous utilisons a été décrit en Section 13.5. Dans l'ensemble, il correspond à l'estimation suivante :

$$P(T|W) \approx P(A|T)P(T)$$

où $P(A|T)$ est le modèle d'ambiguïté lexicale, qui permet à l'analyse syntaxique de garder une référence aux mots.

Dans la même section, nous avons eu l'occasion de mentionner que le modèle de langue a déjà été représenté sous la forme de machines à états finis (Tzoukermann & Radev 1996, Kempe 2000). Le concept n'est donc pas nouveau. Cependant, dans les modèles présentés, une forme lexicale et sa catégorie sont représentées dans les machines sous la forme d'une seule valeur, la classe d'ambiguïté lexicale correspondante. Sous cette forme, le modèle syntaxique et le modèle lexical sont donc estimés dans un même élan.

Pourtant, l'estimation du modèle lexical, indépendamment du modèle syntaxique, est la condition *sine qua non* pour conserver un modèle flexible, c'est-à-dire capable, comme notre modèle d'ambiguïté lexicale, de choisir le mode d'estimation le plus adéquat en fonction du mot.

Principe. Nous nous fondons sur le fait que notre bibliothèque est définie sur le semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$, et sur le fait que le modèle de langue peut être vu comme une somme de logarithmes négatifs :

$$-\log P(T|W) \approx -\log P(A|T) + -\log P(T)$$

pour proposer de représenter le modèle de langue sous la forme d'une composition de deux machines pondérées :

$$-\log P(T|W) \approx M_{P(A|T)} \circ M_{P(T)} \tag{15.5.1.1}$$

Il est donc maintenant envisageable de dissocier les deux modèles, tout en les représentant à l'aide de machines à états finis.

C'est ainsi que nous avons *extrait* le modèle lexical de l'analyse syntaxique, pour l'estimer *au cours de l'analyse morphologique*. A la sortie de l'analyse morphologique, la machine qui représente la phrase est donc pondérée par le modèle lexical uniquement. Cette machine est ensuite composée avec la machine représentant le modèle syntaxique seul.

15.5.1.1 Le modèle lexical

Le nouveau modèle lexical est fortement inspiré de l'ancien, mais y ajoute une estimation adaptée aux OOVs. Le modèle d'ambiguïté lexicale dans son ensemble peut donc être reformulé comme suit :

$$P(A|T) = \prod_{i=1}^{m} p(a_i|t_i) \tag{15.5.1.2}$$

où $p(a_i|t_i)$ est défini comme suit :

$$p(a_i|t_i) = \begin{cases} p(IV_i|t_i) & \text{si } w_i \text{ est IV,} \\ p(OOV_i|t_i) & \text{sinon} \end{cases} \tag{15.5.1.3}$$

Dans ce modèle, $p(IV_i|t_i)$ correspond à l'ancien modèle d'ambiguïté

$$p(IV_i|t_i) = \begin{cases} p(w_i|t_i) & \text{si } w_i \text{ a été observé sur le corpus,} \\ p(c_i|t_i) & \text{si } w_i \in c_i \text{ et } c_i \text{ a été observé sur le corpus,} \\ \frac{1}{N} & \text{sinon} \end{cases} \tag{15.5.1.4}$$

Le modèle $p(OOV_i|t_i)$, réservé aux OOVs, demande quelques explications qui sont données au cours de la description de l'analyse morphologique réalisée sur les OOVs.

Le modèle lexical est donc divisé entre deux machines pondérées : l'une réalise l'analyse morphologique des IVs, et l'autre, celle des OOVs.

15.5.1.2 Le modèle syntaxique

Un tri-gramme est très facile à modéliser à l'aide d'un automate pondéré. La Figure 15.12 en illustre le principe :

1. Les symboles de l'automate sont évidemment les catégories syntaxiques. A ces symboles s'ajoute la catégorie `EOS` [6], qui représente la fin de la phrase.
2. Chaque état de l'automate correspond à un historique distinct. Pour représenter un modèle tri-gramme, il faut donc autant d'états qu'il y a de bi-grammes possibles. De chaque bi-gramme (A, B) et pour chaque catégorie C, part une transition étiquetée C, pondérée par $p(C|A, B)$ et qui atteint l'état correspondant à l'historique suivant, (B, C).
3. Un état supplémentaire, l'état initial, représente le début de la phrase et gère l'initialisation du modèle. Appelons cet état `BOS` [7]. De cet état part une transition pour chaque catégorie A. La transition, étiquetée A et pondérée par la probabilité de A en début de phrase, $P(A|\texttt{BOS})$, joint l'état correspondant à l'uni-gramme en début de phrase, $(\texttt{BOS}, A)$.
4. Les seuls états de l'automate qui sont finaux sont ceux qui correspondent aux bi-grammes terminés par la catégorie `EOS` : $(A, \texttt{EOS})$, $(B, \texttt{EOS})$, $(C, \texttt{EOS})$, etc.

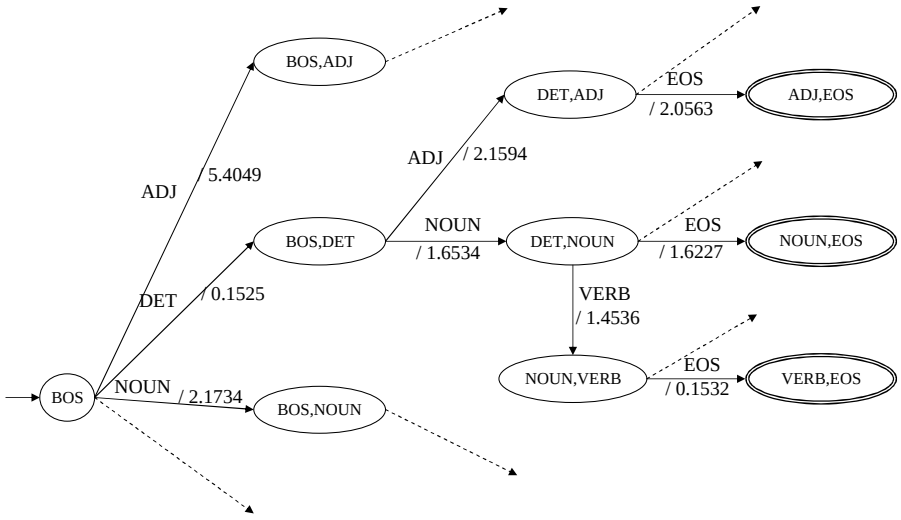


FIG. 15.12: Modèle tri-gramme sous la forme d'un automate pondéré

Compilation. La modélisation décrite ci-dessus pourrait être mise en œuvre manuellement, en construisant pas à pas l'ensemble des états nécessaires. Le modèle peut cependant

[6] *End Of Sentence.*
[7] *Begin Of Sentence.*

être généré sous la forme d'un langage et de règles de réécriture, à l'aide d'Ovide. La Figure 15.13 propose un extrait du fichier de règles.

On constate que le fichier commence par la définition d'une classe de catégories, qui inclut toutes les catégories, sauf `EOS`. Le langage accepté est ensuite décrit, et se limite à toute suite de catégories suivie de `EOS`.

Viennent ensuite les règles de réécriture. Ovide permet, entre autres, la description de règles qui attribuent simplement un poids ou une distance à une séquence de symboles d'un langage, sans réécriture des symboles eux-mêmes. C'est le cas des règles décrites ici, qui se divisent en trois parties :

1. La première partie assure la pondération des bi-grammes initiaux, c'est-à-dire ceux qui estiment la probabilité de chaque catégorie A en début de phrase : $p(A|\texttt{BOS})$.
2. La seconde partie gère la pondération des tri-grammes initiaux, c'est-à-dire ceux qui estiment la probabilité d'un couple de catégories (A, B) en début de phrase : $p(B|\texttt{BOS}, A)$.
3. La troisième partie concerne les tri-grammes qui n'apparaissent pas en début de phrase, c'est-à-dire ceux qui estiment la probabilité d'un triplet de catégories (A, B, C) : $p(C|A, B)$.

De ces n-grammes, seuls les n-grammes terminés par la catégorie `EOS` sont finaux. Ceci est dû au langage défini, qui n'accepte que les séquences de symboles terminées par `EOS`.

La compilation de ces règles par Ovide produit un WFSA, dont le langage est équivalent à celui du WFSA construit manuellement. Ce modèle, qui estime 216 000 tri-grammes, prend 1,6 Mo sous la forme d'une machine à états finis minimisée.

15.5.1.3 Ajout d'un modèle « syntagmatique »

Un modèle de langue présente de nombreuses qualités, telles que la légèreté et la robustesse, mais est par nature incapable de gérer les dépendances de longues distances. Or, de nombreux cas non gérés jouxtent la fenêtre d'analyse du n-gramme. Par exemple, dans « *je n'en veux plus* », le *n'* est juste à la frontière de la fenêtre d'analyse du tri-gramme qui évalue *plus*. Cette information est cependant inaccessible à l'analyse, qui préfère étiqueter *plus* `ADVDEG` [8] plutôt que `ADVN` [9]...

Afin de gérer ce type de structures relativement fréquentes et, somme toute, fort simples, nous avons décidé d'augmenter le modèle syntaxique, en y ajoutant un modèle syntagmatique $P(G)$:

$$P(T^G) = P(T)\,P(G) \tag{15.5.1.5}$$

L'objectif de ce nouveau modèle est de favoriser, au cours de l'analyse, les suites de catégories qui constituent des *syntagmes*. Les syntagmes que nous identifions sont fort simples. Dans l'ensemble, nous tâchons principalement de repérer ce que nous appellerons

[8] Adverbe d'intensité.
[9] Adverbe de négation.

```
[CLASSIN]
  CAT       [ADJ-VERB]

[LANGIN]
  <CAT>* EOS

[RULE]
  # Bi-grammes initiaux : p(CAT|BOS)
  ADJ : : ^ _ / 5.4049
  DET : : ^ _ / 0.1525
  ...
  # Tri-grammes initiaux : p(CAT2|BOS,CAT1)
  ADJ NOUN : : ^ _ / 7.4359
  DET ADJ : : ^ _ / 2.1594
  DET NOUN : : ^ _ / 1.6534
  ...
  # Autres tri-grammes : p(CAT3|CAT1,CAT2)
  ADJ NOUN VERB / 3.1779
  DET ADJ NOUN / 1.1130
  DET NOUN VERB / 1.4536
  ...
  DET ADJ EOS / 2.0563
  DET NOUN EOS / 1.6227
  NOUN VERB EOS / 0.1532
  ...
```

FIG. 15.13: Fichier Ovide : modèle tri-gramme

des *groupes verbaux*. Le groupe verbal est ici simplement un *verbe composé*, fait d'un verbe conjugué et d'un satellite (participe passé, infinitif), qui peut inclure les adverbes et les pronoms personnels qui l'entourent. Par exemple, nous reconnaissons les groupes verbaux suivants :

- `ADVN <PRONPER>? VERB ADVN` : *ne veux plus*, *ne le veux plus*, *n'en veux plus*, etc.
- `ADVN <PRONPER>? AUX ADVN ADV? <SATELLITE>` : *n'ai pas voulu*, *ne l'ai pas voulu*, *ne l'ai pas vraiment voulu*, etc.
- `AUX TIRET? EUPHO TIRET? PRONPERSJ <SATELLITE>` : *a-t-il voulu*, etc.

Les syntagmes repérés sont donc fort simples et ne méritent probablement pas cette dénomination abusive.

Compilation du modèle. Afin de favoriser les suites de catégories qui constituent des syntagmes, nous proposons un modèle qui attribue un poids défavorable à toute catégorie *qui n'a pas été intégrée dans un syntagme.*

Ceci peut être obtenu à l'aide d'un fichier Ovide de règles très simples, que nous illustrons en Figure 15.14. La section des classes définit un marqueur. Il s'agit en fait d'un masqueur, sur lequel les catégories des syntagmes acceptés sont projetées. La dernière règle du fichier, quant à elle, attribue un poids à toutes les catégories rencontrées. Le masqueur empêche la règle de s'appliquer sur les catégories d'un syntagme, ce qui favorise le syntagme par rapport à une suite de catégories non masquées.

```
[CLASSIN]
  SYNTMASK   &1
  CAT        [ADJ-VERB]
  PRONPER    ((PRONPER PRONPERCD)|(PRONPERCD PRONPERCI*)|(PRONPERCI))
  SATELLITE  (INFINIT|PARTPASSE)
[RULE]
  ADVN <PRONPER>? VERB ADVN → <SYNTMASK>
  ADVN <PRONPER>? AUX ADVN ADV? <SATELLITE> → <SYNTMASK>
  AUX TIRET? EUPHO TIRET? PRONPERSJ <SATELLITE> → <SYNTMASK>
  ...
  <CAT> / 1.5
```

FIG. 15.14: Fichier Ovide : modèle syntagmatique

On notera cependant que la sortie de la machine compilée avec Ovide ne présente plus les catégories qui ont été rassemblées en syntagme. Pour remédier à ce problème, on projette l'*entrée* de ce FSM [10].

Utilisation du modèle. Etant donné que le modèle $P(T^G)$ peut être vu comme une somme de logarithmes négatifs :

$$-\log P(T^G) = -\log P(T) + -\log P(G)$$

nous pouvons le représenter sous la forme d'une composition de deux machines pondérées :

$$-\log P(T^G) = M_{P(T)} \circ M_{P(G)} \tag{15.5.1.6}$$

Le modèle de langue, dans son ensemble, correspond dès lors à :

$$-\log P(T|W) \approx M_{P(A|T)} \circ M_{P(T^G)}$$

[10] Ceci peut également être réalisé avec Ovide, au cours de la compilation du modèle. Nous renvoyons le lecteur à la documentation du compilateur, en Annexe B.

Notons cependant qu'en cours de traitement, les deux parties du modèle syntaxique augmenté sont appliquées successivement :

$$-\log P(T|W) \approx M_{P(A|T)} \circ (M_{P(T)} \circ M_{P(G)}) \tag{15.5.1.7}$$

15.5.1.4 Analyse syntaxique

Lorsque le processus entame l'analyse syntaxique, chaque token lexical est déjà pondéré par le modèle lexical. Etant donné que les tokens de type différent ont été traités séparément par le processus, la première étape de l'analyse consiste à construire un seul FSM S représentant la phrase.

Ce FSM S est ensuite composé avec le modèle syntaxique augmenté :

$$S \circ (M_{P(T)} \circ M_{P(G)})$$

Le résultat de cette composition est un FSM S', pondéré à la fois par le modèle lexical et par le modèle syntaxique augmenté. L'analyse se termine par le calcul du meilleur chemin de S', qui est retourné au processus général.

15.5.2 Gestion de la casse

Le modèle de casse concerne exclusivement les OOVs. Il ne s'agit pas, à proprement parler, d'une analyse *linguistique*, mais le modèle ne s'inscrit pas non plus dans la correction orthographique, étant donné qu'il ne remet pas en cause la qualité du texte. Son objectif est d'interdire tout traitement supplémentaire dès qu'un nom propre ou un acronyme est repéré.

Le modèle est très simple et correspond au fichier Ovide présenté en Figure 15.15. Le principe est d'attribuer la catégorie *acronyme* à toute forme qui est complètement en majuscule, et la catégorie *nom propre* à toute forme qui commence par une majuscule.

```
[CLASSIN]
  MIN      [a-z àâäèéêëìîïòôöùûüç ´]
  MAJ      [A-Z ÀÂÄÈÉÊËÌÎÏÒÔÖÙÛÜÇ ´]
  LETTER   [<MIN><MAJ>]
[LANGIN]
  <MAJ><LETTER>*
[RULE]
  \x00?→ GND NND 3rd ACRONYM :: ^<MAJ>+ _ $ / 1
  \x00?→ GND NND 3rd PROPERNAME :: ^<MAJ><LETTER>+ _ $ / 2
```

FIG. 15.15: Fichier Ovide : modèle de casse

Une forme complètement en majuscule se voit donc attribuer les deux catégories. Dans ce cas, nos règles avantagent l'acronyme en lui attribuant un poids moins important. L'analyse des corpus à notre disposition montre en effet que les formes en majuscule sont majoritairement des acronymes. Il arrive cependant que des noms propres soient complètement en majuscule. C'est la raison pour laquelle ces formes reçoivent les deux catégories : le modèle syntaxique peut ainsi choisir l'analyse la plus probable en fonction du contexte.

Le langage accepté par le modèle se limite aux formes qui commencent par une majuscule. Ceci permet au système de réaliser un test : si la composition de l'OOV avec le modèle de casse donne un résultat, le traitement s'arrête là : l'OOV commence au moins par une majuscule et a été étiqueté. Dans le cas contraire, d'autres traitements peuvent être réalisés : correction et analyse morphologique.

15.5.3 Analyse morphologique

Nous rappelons que l'analyse morphologique est principalement une analyse flexionnelle, qui détermine, pour une forme donnée, les triplets {nature, classe flexionnelle, désinence} qui lui correspondent. Cette analyse, en soi, concerne toutes les formes lexicales, quel que soit le token. Le véritable lieu de l'analyse morphologique est cependant le token lexical et certaines parties des URIs, dont les formes peuvent admettre plusieurs analyses.

Dans la première version de l'analyseur morphologique (cf. Section 13.4.2.2), un trie de flexion permettait de détecter toutes les flexions potentielles dans une forme, mais un processus de validation ne conservait de ces flexions que celles à partir desquelles il était possible de reconstruire un lemme valide. L'effort d'analyse morphologique était dès lors inutile, puisque les OOVs, qui auraient dû en profiter, en sortaient sans analyse et recevaient *ex nihilo* les quatre catégories ouvertes (`ADJ`, `ADV`, `NOUN` et `VERB`), sans aucune distinction.

Ce constat nous a conduit à poser les deux hypothèses suivantes, qui établissent une distinction fondamentale entre IVs et OOVs :

Hypothèse 15.5.1 (Dictionnaire d'IVs)**.** *En synthèse, les formes lexicales connues du système ne nécessitent pas d'analyse morphologique. Un dictionnaire suffit amplement, pour autant que la structure représentant le dictionnaire soit légère et autorise un accès rapide. Les machines à états finis respectent ces contraintes.*

Hypothèse 15.5.2 (Analyse morphologique des OOVs)**.** *En synthèse, les formes lexicales inconnues sont les seules formes à nécessiter une véritable analyse morphologique. Cette analyse est nécessaire afin d'orienter au mieux le choix de la catégorie à attribuer à la forme lors de l'analyse syntaxique.*

Dans la nouvelle analyse morphologique, les IVs reçoivent dès lors leurs analyses sur base de la simple consultation d'un dictionnaire. Les OOVs, par contre, sont soumis à une véritable analyse morphologique, pour autant qu'il ne s'agisse pas de noms propres ou d'acronymes, et qu'aucun candidat n'ait été trouvé par le système de correction.

15.5.3.1 Analyse des IVs

Base. Chaque token possède un dictionnaire des formes qu'il admet. Dans le cas des tokens lexicaux, ce dictionnaire a été généré à partir du lexique de lemmes et du lexique de flexions décrits en Section 13.4.

Chaque dictionnaire est représenté sous la forme d'un fichier Ovide, qui est compilé en machine à états finis. Comme nous l'avons mentionné en Section 6.2, Ovide autorise la modélisation d'un dictionnaire, qui est déclaré par le mot-clef `DICIN` (cf. Figure 15.16). Les lignes du fichier, de ce fait, ne sont pas interprétées comme des règles de réécriture, mais comme des entrées du dictionnaire. Elles sont donc ajoutées petit-à-petit dans la machine, qui prend la forme temporaire d'un arbre déterministe, minimisé à la fin du processus.

```
[DICIN]
  couvent → m,sp,-s+,-+ Msc Sg PND NOUN / 1.2228
  couvent → 6 GND Pl 3rd VERB / 0.8078
  couvents → m,sp,-s+,-+ Msc Pl PND NOUN / 0
  ...
```

FIG. 15.16: Fichier Ovide : dictionnaire pondéré

On constate sur la Figure 15.16 qu'une forme lexicale est séparée de son analyse morphologique par une flèche. Celle-ci indique que l'analyse ne figure que sur la sortie de la machine, tandis que la forme lexicale se trouve en entrée *et* en sortie. Le FSM correspondant est donc de la forme de celui présenté en Figure 15.17 : en entrée, le FSM n'accepte que des formes fléchies (*couvent* et *couvents*) ; en sortie, chaque forme fléchie est associée à une ou plusieurs analyses pondérées.

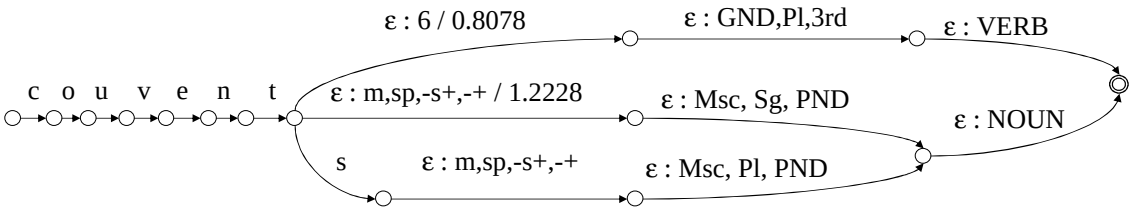


FIG. 15.17: Dictionnaire sous la forme d'un transducteur pondéré

En français, le dictionnaire des IVs lexicaux représente 441 780 formes, compilées sous la forme d'un FSM de 2,3 Mo.

Langage. Quel que soit le token traité, l'algorithme de segmentation conserve toujours les formes connues dans un seul FSM (cf. Section 15.3). L'objectif est de limiter le nombre d'opérations réalisées sur des formes qui ne présentent aucune difficulté.

Les dictionnaires ont donc dû être adaptés, afin qu'ils acceptent en entrée une suite de plusieurs formes. Pour ce faire, chaque fichier Ovide qui représente un dictionnaire D est inclus dans un autre fichier Ovide, dont l'objectif est de générer le langage L qui correspond à une ou plusieurs formes de D séparées par le seul séparateur accepté, l'espace (cf. Section 15.4.1) :

$$L = D \ (\text{" "} D)^*$$

où " " représente l'espace.

La Figure 15.18 en donne un exemple : la section `INCLUDE` inclut le dictionnaire des formes lexicales, *Lex_ inc*, et la section `COMPILE` décrit le langage accepté.

```
[INCLUDE]
  Lex_inc
[COMPILE]
  @Lex_inc (" " @Lex_inc)*
```

FIG. 15.18: Fichier Ovide : langage à partir d'un dictionnaire

Note 15.5.1. Afin de conserver une distinction claire entre ce type de machines et les modèles plus complexes, le langage compilé sur un dictionnaire est qualifié lui-même de *dictionnaire*.

Application. La recherche d'une suite de formes, W, dans un dictionnaire *Lexicon* est toujours réalisée par composition relâchée, de manière à gérer les différences de casse et d'accentuation :

$$w \circ Match \circ Lexicon$$

Le format du résultat obtenu correspond à l'étape *2.a* de la construction du format compatible avec l'analyse syntaxique (cf. Figure 15.4, Section 15.4.1). Pour passer à l'étape *2.b*, la sortie du résultat est projetée.

15.5.3.2 Analyse des OOVs

Nous rappelons qu'un OOV n'est soumis à l'analyse morphologique que si le modèle de casse et le système de correction n'ont donné aucun résultat.

Afin de s'intégrer dans le modèle lexical présenté, cette analyse doit en outre être pondérée. Etant donné un mot i qui est OOV, ce modèle peut, selon la règle de Bayes, se formuler comme suit :

$$p(OOV_i|t_i) = \frac{p(t_i|OOV_i)}{p(t_i)} \tag{15.5.3.1}$$

Dans ce modèle, la question est de déterminer *comment* $p(t_i|OOV_i)$ peut être estimé.

Réflexions. Notre modèle d'analyse morphologique se base sur les réflexions suivantes :

1. Sur la base d'une information flexionnelle de qualité, le nombre des catégories ouvertes peut être augmenté, de manière à inclure les sous-catégories verbales utilisées dans eLite : `INFINIT`, `PARTPASSE` et `PARTPRES`.
2. L'analyse flexionnelle des OOVs doit se contenter de la détection de flexions potentielles, sans possibilité de validation.
   (a) Or, sans phase de validation, le nombre de flexions détectées dans une forme lexicale peut être conséquent. Par exemple, l'ensemble des suffixes détectés dans la forme *mènent* est {∅, *-t -nt -ent*, *-nent*, *-ènent*}, alors que le seul suffixe validé est *-ènent*.
   (b) On pourrait envisager de ne conserver que la plus longue flexion détectée. Cependant, une flexion, même longue, pourrait être détectée à tort. Prenons le mot *vent* et admettons que ce mot soit absent de notre dictionnaire. La détection de la flexion *-ent* serait erronée. La système ne peut donc se contenter d'une analyse unique.

   L'analyse morphologique d'un OOV ne peut dès lors être strictement flexionnelle, étant donné qu'aucune phase de restriction n'est envisageable, et que la détection d'une seule flexion est fort restrictive.
3. Le dictionnaire de formes fléchies généré à partir de notre lexique de lemmes compte 441 780 formes. La couverture des flexions de la langue est donc certainement de qualité. Nous faisons dès lors l'hypothèse qu'une terminaison qui n'apparaîtrait pas un minimum de fois dans ce lexique serait négligeable.
4. Etant donné que l'analyseur morpho-syntaxique autorise l'estimation de distances et de probabilités, il peut être intéressant que les analyses morphologiques d'un OOV soient pondérées.

Décisions. Ces réflexions nous conduisent aux décisions suivantes :

1. Le modèle peut proposer `INFINIT`, `PARTPASSE` et `PARTPRES`, pour les terminaisons qui admettent ces analyses dans le lexique.
2. En l'absence d'une possibilité de valider les analyses flexionnelles détectées,
   (a) Nous proposons que l'analyse délaisse la notion stricte de *flexion* au profit de la *plus longue terminaison possible*, de manière à inclure dans l'estimation morphologique des affixes non flexionnels, voire des radicaux. Cependant, afin de maintenir la taille du FST dans des dimensions raisonnables, nous fixons la longueur maximale de la terminaison à 6 caractères.
   (b) Etant donné qu'une terminaison, même longue, peut être détectée à tort, le système se doit d'attribuer systématiquement l'ensemble des catégories ouvertes, qui reste limité à {`ADJ`, `ADV`, `NOUN`, `VERB`}. Les analyses `INFINIT`, `PARTPASSE` et

`PARTPRES` ne font donc partie de l'analyse que si la terminaison les autorise explicitement.

3. Une terminaison n'est intégrée au modèle que si son nombre d'occurrences est supérieur à un seuil donné. Expérimentalement, nous avons fixé ce seuil à 30 occurrences.
4. Afin d'obtenir une pondération consistante de l'ensemble des analyses morphologiques, nous proposons de baser la pondération d'une analyse sur le lexique de formes fléchies. Ce lexique offre en effet l'avantage de présenter la *totalité* des paradigmes des lemmes connus de nos lexiques. La pondération évalue donc simplement la probabilité, dans le lexique, d'une catégorie t étant donné une terminaison f :

$$p(t|f) = \frac{c(t,f)}{c(f)}$$

Cette évaluation doit cependant être lissée, étant donné que les catégories qui ne sont pas proposées par l'analyse, mais ajoutées à l'analyse, doivent également être pondérées. Nous avons employé le lissage additif (cf. Section 13.5.3.3) :

$$p_{ADD}(t|f) = \frac{\delta + c(t,f)}{\delta + c(f)}$$

où δ vaut $0,1$. Etant donné un mot i qui est OOV et qui finit par la terminaison f, le modèle lexical se reformule donc comme suit :

$$p(OOV_i|t_i) = \frac{p_{ADD}(t_i|f)}{p(t_i)} \tag{15.5.3.2}$$

Construction des règles. L'algorithme de construction des règles d'analyse morphologique est présenté en Pseudocode 34. A partir du dictionnaire des formes fléchies, la première étape crée une liste exhaustive des terminaisons existantes, où chaque terminaison correspond au plus aux 6 derniers caractères des formes lexicales (lignes 1–5). La forme *apparaîtra*, par exemple, sera tronquée en *raîtra*.

L'algorithme ne retient ensuite que les terminaisons d'au moins 30 occurrences (lignes 6–9). Pour ce faire, le principe est de supprimer de la liste une terminaison de moins de 30 occurrences, de la tronquer d'un caractère, et d'ajouter ou de mettre à jour la terminaison tronquée dans la liste. Ceci explique que les terminaisons soient traitées dans l'ordre décroissant de leur longueur, afin que toutes les terminaisons de longueur n qui sont supprimées puissent participer à l'estimation des terminaisons de longueur $n-1$. Le processus pourrait par exemple supprimer, créer et mettre à jour successivement *raîtra* → *aîtra* → *îtra* → *tra*, s'arrêtant lorsque le nombre d'occurrences est suffisant.

Lorsque l'ensemble des terminaisons est fixé, le fichier de règles est créé (lignes 10–19). Les terminaisons sont une fois encore traitées dans l'ordre décroissant de leur longueur. Ceci est une conséquence de l'un des principes fondamentaux des règles de réécriture, qui veut que toute règle doit précéder dans le fichier une autre règle dont elle est un cas particulier,

1 : **Pour chaque** 4-uplet {forme, catégorie, classe, désinence} du lexique,
2 : La terminaison vaut au plus les 6 derniers caractères de la forme.
3 : Mémoriser la terminaison **ou** Mettre son compteur global à jour.
4 : Mémoriser la paire {terminaison, catégorie} **ou** Mettre son compteur à jour.
5 : Mémoriser le 4-uplet {terminaison, catégorie, classe, désinence}.
6 : **Pour chaque** terminaison traitée en taille décroissante,
7 : **Si** son compteur global est inférieur à 30,
8 : Supprimer le premier caractère.
9 : Mettre à jour les données de la nouvelle terminaison obtenue.
10 : **Pour chaque** terminaison *End* traitée en taille décroissante,
11 : **Si** son compteur global est supérieur ou égal à 30,
12 : **Pour chaque** catégorie T associée à la terminaison,
13 : Calculer $P_{ADD}(T, End)$.
14 : Ajouter au fichier de règles une règle pour le couple (T, End) qui tient compte de toutes les paires {classe, désinence} qui sont associées à la catégorie pour cette terminaison.
15 : **Si** des catégories ouvertes n'ont pas été traitées,
16 : Créer l'ensemble T_O des catégories ouvertes non traitées.
17 : Calculer $P_{ADD}(T_O, End)$.
18 : Ajouter au fichier de règles une règle pour le couple (T_O, End) associé à la *désinence neutre*.
19 : Ajouter au fichier de règles une règle pour l'ensemble des catégories ouvertes associées à la *désinence neutre*.

Pseudocode 34: Construction des règles d'analyse morphologique d'un OOV

afin qu'il n'y ait pas de phénomène de masquage non désiré. En l'occurrence, une terminaison longue est bien souvent un cas particulier d'une terminaison plus courte, et notre objectif est de détecter uniquement la plus longue terminaison possible.

Les règles produites pour une terminaison donnée (lignes 12–18) tiennent compte des contraintes suivantes :

1. L'association d'une catégorie à une terminaison est pondérée. Il faut donc une règle par catégorie.
2. Pour éviter le phénomène de masquage décrit précédemment, il est dès lors nécessaire que seule la dernière règle associée à une terminaison soit obligatoire, les autres étant optionnelles.
3. Dans une règle, chaque catégorie doit être associée aux classes et aux désinences de la terminaison.
4. Si des catégories ouvertes n'ont pas été observées, une règle doit les concerner. Ces catégories sont dans ce cas associées à la désinence neutre `GND`, `NND`, `PND`.

Les règles sont donc de la forme :

$$\texttt{\textbackslash x00 ?}\rightarrow \texttt{T}_1\ \texttt{C}_1\ \texttt{Gn}_1\ \texttt{Nb}_1\ \texttt{Prs}_1\ \texttt{::}\ \texttt{End\ \_\ \$\ /}\ w$$
$$\texttt{\textbackslash x00 ?}\rightarrow \texttt{T}_2\ \texttt{C}_2\ \texttt{Gn}_2\ \texttt{Nb}_2\ \texttt{Prs}_2\ \texttt{::}\ \texttt{End\ \_\ \$\ /}\ w$$
$$\ldots$$
$$\texttt{\textbackslash x00}\ \rightarrow \texttt{T}_n\ \texttt{C}_n\ \texttt{Gn}_n\ \texttt{Nb}_n\ \texttt{Prs}_n\ \texttt{::}\ \texttt{End\ \_\ \$\ /}\ w$$

où C_i est la classe flexionnelle, Gn_i est le genre, Nb_i est le nombre et Prs_i, la personne, pour $1 \leq i \leq n$. Par exemple, la terminaison *-vent* produit les règles suivantes :

```
\x00 ?→ NOUN m,sp,-+,-+ Msc Sg PND :: vent _ $ / 4.7751
\x00 ?→ VERB (6|9a|80) GND Pl 3rd :: vent _ $ / 0.0546
\x00 → (ADJ|ADV) GND NND PND:: vent _ $ / 11.5565
```

A partir du lexique de formes fléchies, l'application de cet algorithme génère 12 957 règles, qui prennent 11,5 Mo sous la forme d'un FST.

Application du modèle. Un mot W à analyser est simplement composé avec le modèle d'analyse morphologique $OOVMorpho$:

$$W \circ OOVMorpho$$

La Figure 15.19 illustre le comportement du modèle sur un mot du lexique, *couvent*. L'analyse lui attribue les quatre catégories ouvertes, mais `VERB` et `NOUN` sont les mieux pondérées, ce qui correspond effectivement au contenu du lexique. La différence de pondération entre `VERB` et `NOUN` est due à la terminaison exclusivement, et non à la forme complète. Le comportement du modèle semble donc sein.

La Figure 15.20 illustre quant à elle l'analyse d'un véritable OOV, *dumpent*, qui est la 3^{re} personne du pluriel du verbe *dumper*, emprunté à l'anglais (*to dump*). Le modèle favorise cette analyse, mais propose également les autres catégories ouvertes.

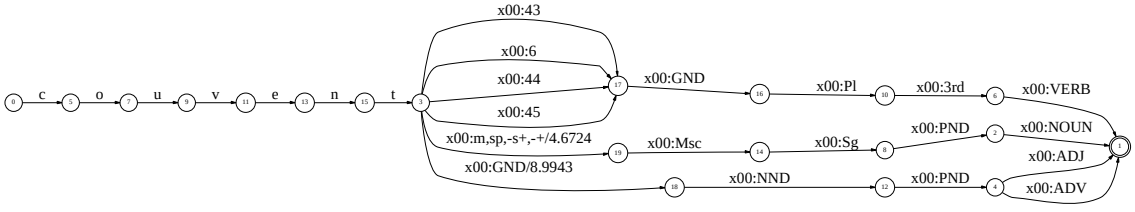


FIG. 15.19: Analyse morphologique : illustration du comportement sain sur un IV

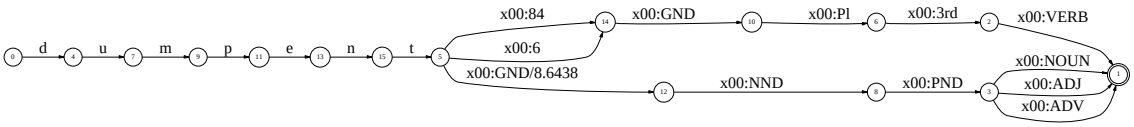


FIG. 15.20: Analyse morphologique d'un véritable OOV

15.5.4 Génération des unités linguistiques

Note 15.5.2. Pour une plus grande clarté de la présentation, les règles présentées en illustration de cette section ne contiennent pas de flexions.

Nous rappelons que, quel que soit le type de token, la phase de génération des unités linguistiques est réalisée lorsque l'analyse morphologique est terminée. Dans la machine à traiter, chaque forme lexicale est donc suivie de son analyse flexionnelle.

Nous rappelons également que, après le traitement, la sortie de la machine ne peut plus contenir *que* des catégories syntaxiques, comme le montre la Figure 15.4, Section 15.4.1. La figure montre également que la catégorie syntaxique *doit être alignée* sur la première nature grammaticale de l'unité linguistique. Ceci facilite considérablement la sauvegarde finale des informations dans la DLS.

Au sein du token lexical, trois cas de figure sont à traiter, selon que les formes lexicales forment ou non un mot composé. Les autres tokens, quant à eux, sont tous soumis à un processus rigoureusement identique.

Les mots composés détachés. La catégorie syntaxique dépend ici directement de la suite de mot et, comme nous l'avons signalé, doit être alignée sur la nature grammaticale de la première forme lexicale du composé. Nous allons illustrer le procédé à l'aide du composé « *vis à vis d'* », qui est une préposition et dont le traitement doit donner le résultat de la Figure 15.21.

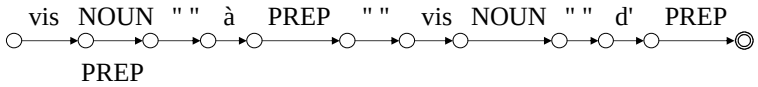


FIG. 15.21: Catégorie syntaxique d'un composé détaché

Pour obtenir ce résultat, une règle doit donc convertir la première nature grammaticale :

`NOUN → PREP :: vis _ " " à PREP " " vis NOUN " " d' PREP`

et une autre règle doit supprimer tous les autres symboles. Cependant, la règle de suppression ne peut être définie, parce que les natures grammaticales et les catégories syntaxiques partagent les mêmes symboles. La règle supprimerait donc trop ou pas assez de symboles...

Pour pallier ce problème, nous avons recours aux marqueurs définis et présentés en Section 6.2.4. Le marqueur nécessaire ici est le *masqueur*, qui est employé dans une règle en lieu et place d'un symbole, de manière à éviter au symbole en question d'être réécrit par une ou plusieurs règles suivantes. La Figure 15.22 illustre l'utilisation du masqueur dans ce cas précis. Dans la section réservée aux classes, un masqueur est défini pour la catégorie PREP, et est employé en lieu et place de cette catégorie dans la règle d'attribution de la catégorie syntaxique. La règle de suppression est dès lors définie sans risque, et supprime tous les symboles ; le masqueur, qui ne fait pas partie de l'alphabet, n'est pas concerné par cette règle. Enfin, une dernière règle convertit le masqueur, sans risque qu'il soit supprimé.

```
[CLASSIN]
  PREPMASK      &1
[RULE]
  NOUN → <PREPMASK> :: vis _ " " à PREP " " vis NOUN " " d' PREP
  . → \x00
  <PREPMASK> → PREP
```

FIG. 15.22: Fichier Ovide : unité linguistique des composés détachés

Les formes lexicales simples et les composés sur les natures. La catégorie syntaxique d'une forme lexicale simple *est* sa nature grammaticale. Le seul traitement à réaliser est donc de supprimer les autres symboles de la sortie. Or, afin de tester la possibilité de recomposer plusieurs formes lexicales en fonction de leurs natures, la première étape est de supprimer, de la sortie de la machine, tous les symboles qui ne sont pas des natures. Cette suppression est le résultat d'une composition avec une machine qui projette tout symbole qui n'est pas une catégorie sur ϵ :

$$[\hat{}\texttt{<CAT>}] \rightarrow \backslash\texttt{x00}$$

où <CAT> est la classe de l'ensemble des natures et des catégories de l'alphabet.

La seconde étape est de composer le résultat de la suppression avec un modèle qui convertit certaines suites de natures en une seule catégorie syntaxique. Les règles sont très simples :

$$\texttt{PREF TIRET NOUN} \rightarrow \texttt{NOUN}$$

et sont déduites automatiquement du lexique de composés sur natures que nous avons présenté en Section 13.4.2.

A la fin de ce processus, tous les composés sur les natures sont détectés, et les formes lexicales simples ne présentent plus, en sortie, que leur catégorie syntaxique.

Les autres tokens. Qu'il s'agisse d'une URI ou d'un autre token, la valeur de l'unité linguistique est déterminée de manière non ambiguë par le token, et l'unité linguistique correspond toujours à un et un seul token. De ce fait, nous nous sommes contenté d'un traitement fort simple, constitué de deux étapes :

1. Nous appliquons sur la machine qui représente le token la méthode `FSM_EpsilonOut`, définie dans notre bibliothèque et qui remplace tous les symboles de la sortie par ϵ.
2. Nous suivons les transitions de la machine depuis l'état initial, jusqu'à ce que nous ayons trouvé la transition correspondant à la première nature grammaticale de l'unité linguistique. Nous modifions alors le symbole de sortie de cette transition, en y copiant la catégorie syntaxique de l'unité.

15.5.5 Les points-clefs de l'analyse morpho-syntaxique

L'analyse morpho-syntaxique que nous proposons est exclusivement gérée à l'aide de machines à états finis. Elle a demandé la scission du modèle de langue entre les deux parties de l'analyse, de manière à conserver une flexibilité totale du modèle lexical.

Les principes de l'analyse morphologique varient selon que la forme est IV ou OOV. L'analyse des IVs consiste en une simple consultation d'un dictionnaire, pondéré par le modèle lexical :

$$p(IV_i|t_i) = \begin{cases} p(w_i|t_i) & \text{si } w_i \text{ a été observé sur le corpus,} \\ p(c_i|t_i) & \text{si } w_i \in c_i, \text{ et que } c_i \text{ a été observé sur le corpus,} \\ \frac{1}{N} & \text{sinon} \end{cases}$$

Les OOVs sont par contre soumis à une véritable analyse morphologique, pour autant que le modèle de casse et le système de correction n'aient donné aucun résultat. L'analyse morphologique des OOVs dépasse les flexions, pour estimer la probabilité de terminaisons plus englobantes, à l'aide du modèle lexical :

$$p(OOV_i|t_i) = \frac{p_{ADD}(t_i|f)}{p(t_i)}$$

Le modèle lexical, dans son ensemble, correspond à l'union :

$$p(a_i|t_i) = \begin{cases} p(IV_i|t_i) & \text{si } w_i \text{ est IV,} \\ p(OOV_i|t_i) & \text{sinon} \end{cases}$$

A la fin du processus, chaque forme lexicale est accompagnée de son analyse flexionnelle, quel que soit le token.

La génération des unités linguistiques se résume généralement à une manipulation très simple, qui réduit simplement la sortie de la machine à la catégorie choisie. Le processus est cependant plus complexe dans le cas de tokens lexicaux, pour lesquels des composés détachés ou des recomposés sur les natures doivent être gérés. A la fin du processus, seules les catégories syntaxiques figurent encore sur la sortie de la machine.

L'analyse syntaxique conclut la chaîne des traitements appliqués à une phrase. Elle correspond désormais au modèle syntaxique augmenté, $P(T^G)$, combinaison du modèle syntaxique classique et d'un nouveau modèle syntagmatique fort simple. Reconstitué par composition, le modèle de langue complet correspond donc à :

$$P(T|W) \approx P(A|T)\, P(T^G)$$

15.6 Les modèles de la correction orthographique

Sur la base des réflexions exposées en Section 15.2.1, l'objectif de notre système de correction est de gérer les OOVs et les erreurs flexionnelles. Cette section aborde séparément ces deux types de correction.

15.6.1 Correction des OOVs

Note 15.6.1. Nous avons eu l'occasion de mentionner qu'un OOV peut être une forme corrompue, mais également un nom propre ou un acronyme, un mot de la langue absent de notre dictionnaire, ou un emprunt à une langue étrangère. Dans cette section, l'OOV fait exclusivement référence à la forme corrompue.

15.6.1.1 Approche générale

En présence d'un OOV, l'objectif du système de correction est de déterminer la liste des mots du dictionnaire qui sont susceptibles d'y correspondre. Une recherche dans le dictionnaire est donc nécessaire. Cette recherche implique la notion de *distance d'édition*.

Dans ce contexte, nous proposons un modèle original, basé sur la notion de *composition filtrée* (cf. Section 15.4.2.2) : chacun de nos modèles de correction est un filtre de composition F, qui se place entre l'OOV et le dictionnaire pour réaliser la distance d'édition :

$$OOV \circ F \circ Lexicon$$

Etant donné que dans le cadre d'un document dactylographié, un OOV est le résultat d'une erreur typographique ou d'une erreur phonétique, nous proposons deux filtres d'édition : un filtre typographique $Typo$, et un filtre phonétique $Phonet$. Ces deux filtres sont appliqués séparément, de sorte que les candidats de la recherche correspondent à l'union de ces compositions filtrées :

$$Cand = ((OOV \circ Typo \circ Lexicon) \mid (OOV \circ Phonet \circ Lexicon))$$

Si des candidats existent, ils sont accompagnés de leurs analyses morphologiques, qui sont pondérées par le système suivant :

$$p(IV|OOV, t_i) = D(OOV, IV) + -\log p(IV|t_i) \tag{15.6.1.1}$$

où $p(IV|t_i)$ est le modèle lexical des IVs, et $D(OOV, IV)$ est la distance d'édition, soit typographique, soit phonétique, qui a permis de trouver le candidat IV. Ceci demande deux commentaires :

1. Contrairement aux systèmes de l'état de l'art (cf. Section 14.4.1), qui calculent une probabilité d'édition à partir des candidats y trouvés par la distance d'édition, nous conservons une *distance*. Deux raisons le justifient. D'une part, nous ne conservons pas l'OOV, mais seulement les candidats trouvés. D'autre part, nous tenons à différencier les candidats sur la base du nombre d'opérations d'édition nécessaires pour les retrouver. L'état de l'art ne permettait pas cette distinction, qui nous semble importante.
2. Nos filtres permettent de trouver des *solutions*, mais ne choisissent en aucune manière *la* correction à retenir. Ce choix est réalisé au cours de l'analyse syntaxique, lors du calcul du meilleur chemin.

Limites d'édition. Notre survol de l'état de l'art a mis en évidence que la correction des OOVs est le plus souvent réduite à la recherche des IVs distants d'une seule opération d'édition. Ce choix, que les auteurs justifient par le fait que 80% des OOVs ne présentent qu'une seule erreur d'édition, est probablement motivé également par l'absence d'un outil autorisant une variation du nombre d'erreurs d'une forme à l'autre. En ce qui nous concerne, nous formulons le postulat suivant :

Postulat 15.6.1 (Limites d'édition)**.** *Le nombre d'erreurs autorisées dans une forme lexicale doit tenir compte de trois facteurs : la longueur de la forme, le type de l'erreur et les limites de la compréhension humaine.*

et nous faisons l'hypothèse suivante :

Hypothèse 15.6.1 (Limites d'édition)**.** *Un filtre de composition est le média idéal pour modéliser aisément les trois facteurs qui influent sur le nombre d'erreurs présentes dans une forme lexicale.*

La conception de nos filtres se fonde sur ce postulat :

1. Le filtre typographique tient compte de la longueur du mot, de l'opération d'édition et de la disposition des touches sur le clavier pour préciser le nombre, le lieu et le type des opérations d'édition acceptées.
2. Le filtre phonétique limite les variations *graphiques* sur la base de contraintes intralexicales.

Les points suivants détaillent les principes et la construction de ces deux filtres.

15.6.1.2 Le filtre typographique

Le principe. Les contraintes dépendantes de la longueur et celles imposées aux opérations d'édition sont modélisées dans deux FSMs séparés. Le FSM *Long* modélise les contraintes de longueur, et le FSM *Edit* modélise les opérations d'édition autorisées.

La recherche typographique d'un OOV dans le dictionnaire correspond donc à la cascade de compositions suivante :

$$OOV \circ (Long \circ Edit) \circ Lexicon$$

La modélisation des FSMs *Long* et *Edit* repose sur l'utilisation des marqueurs définis et présentés en Section 6.2.4. Dans le principe, des marqueurs sont introduits par *Long* dans l'OOV en fonction de sa longueur. Ces marqueurs sont des *déclencheurs*, qui indiquent à *Edit* les endroits où les règles qu'il décrit peuvent être appliquées.

Les contraintes sur la longueur. Nous avons défini des intervalles de longueur qui autorisent un certain nombre d'opérations d'édition :

1. De 1 à 5 caractères : 1 opération.
2. De 6 à 10 caractères : 2 opérations.
3. De 11 à 15 caractères : 3 opérations.
4. Plus de 15 caractères : 4 opérations.

La Figure 15.23 illustre le procédé, par un extrait de quelques règles simples, permettant l'insertion des marqueurs désirés.

Dans la section des classes sont définis des marqueurs de longueur (`L1` à `L4`) et un marqueur d'édition (`ED`). La classe `CHAR` est simplement définie pour faciliter la lecture, et correspond aux caractères pouvant appartenir à un token lexical.

Dans la section des règles, les premières règles insèrent un marqueur qui identifie la longueur du langage correspondant. Par exemple, un mot de 1 à 5 caractères est marqué `L1`. La dernière règle insère après tout symbole un marqueur d'édition.

Vient ensuite une section qui décrit le langage accepté en sortie du transducteur. Ce langage est en fait l'union des différentes expressions de la section. Dans notre exemple, la première expression accepte toute suite de caractères sans marqueurs, la seconde accepte un marqueur d'édition, quelle que soit la longueur du mot, et la troisième accepte deux marqueurs, pour autant que le mot contienne au moins 6 caractères.

A la fin du processus de compilation, le transducteur représentant les règles est composé avec l'automate représentant le langage. Le résultat ne contient plus que les règles qui respectent les contraintes décrites par le langage.

Les véritables règles de notre filtre sont un peu plus complexes que celles présentées, mais suivent le même principe. Nous employons en réalité un marqueur spécifique pour

chaque opération d'édition, de manière à pouvoir contraindre, de manière plus fine, les opérations d'édition qui peuvent apparaître simultanément dans une forme. Dans l'ensemble, le langage refuse plusieurs opérations successives. En outre, insertion, suppression et transposition ne peuvent se produire en même temps. Nous employons 16 règles, dont la compilation donne un FST de 622 Ko.

La Figure 15.24 donne un exemple d'application des règles de la Figure 15.23 sur le mot *vent*. Ce mot ne compte que 4 lettres et est de ce fait identifié par le marqueur `L1`, qui n'autorise qu'un seul marqueur d'édition par chemin.

```
[CLASSIN]
  L1      &1
  L2      &1
  L3      &1
  L4      &4
  LALL    [<L1>-<L4>]
  ED      &5
  CHAR    .
[RULE]
  \x00 → <L1> : : ^ _ <CHAR>{1,5}$
  \x00 → <L2> : : ^ _ <CHAR>{6,10}$
  \x00 → <L3> : : ^ _ <CHAR>{11,15}$
  \x00 → <L4> : : ^ _ <CHAR>
  \x00 ?→ <ED> : : <CHAR> _
[LANGOUT]
  <LALL> <CHAR>*
  <LALL> <CHAR>* <ED> <CHAR>*
  [<L2>-<L4>] <CHAR>* <ED> <CHAR>+ <ED> <CHAR>*$
  ...
```

FIG. 15.23: Fichier Ovide : limites sur les opérations d'édition

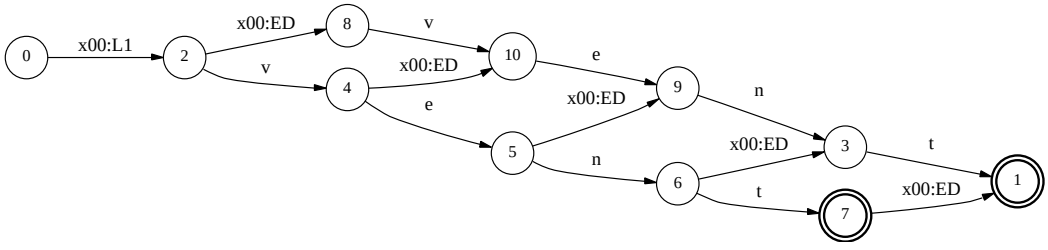


FIG. 15.24: Distance d'édition : limites sur les opérations à l'aide de marqueurs

Les contraintes sur les opérations. Les contraintes appliquées aux opérations se basent sur quelques réflexions que nous présentons ci-dessous. Ces réflexions sont illustrées par des règles qui ne contiennent pas de marqueurs, pour des raisons de lisibilité :

1. L'*insertion* d'un caractère dans l'OOV ne doit dépendre d'aucun contexte. En effet, l'utilisateur peut ne pas avoir appuyé suffisamment fort sur une touche, quelle que soit sa place sur le clavier. Il faut dès lors une seule règle d'insertion :

   `\x00 ? → <CHAR> / 2`

   où `<CHAR>` est la classe de l'ensemble des caractères acceptés dans un token lexical.

2. La substitution d'un caractère pour un autre est *a priori* fort dépendante de la position des touches sur le clavier : l'utilisateur risque plus facilement de substituer deux caractères voisins. Nous restreignons de ce fait la substitution aux touches directement voisines sur le clavier. Par exemple,

   `b ? → [fghvn] / 1`

3. La *suppression* d'un caractère est également fort dépendante de la position des touches sur le clavier : l'utilisateur risque plus facilement d'accrocher les touches du clavier qui sont voisines. Nous favorisons de ce fait la *suppression* lorsque les lettres concernées correspondent à des touches contiguës sur le clavier. Par exemple,

   ```
   b ? → \x00 :: _ [fghvn] / 2
   b ? → \x00 :: [fghvn] _ / 2
   b ? → \x00 / 3
   ```

   Les deux premières règles favorisent la suppression de b lorsqu'il est précédé ou suivi de f, g, h, v ou n, qui lui sont contiguës sur le clavier. La troisième règle ne s'appliquera que si le contexte est différent.

4. La transposition peut évidemment se produire entre toute paire de caractères, quelles que soient leurs places sur le clavier. Par exemple,

   `ab ? → ba / 2.5`

La Figure 15.25 donne un exemple de règles concernant le caractère b et son contexte. Toute règle contient un marqueur d'édition, qui sert de *déclencheur* : la règle ne s'applique à une forme que si celle-ci présente le marqueur adéquat. En outre, l'application de la règle supprime le marqueur, afin qu'aucune autre règle ne puisse s'appliquer par la suite.

On constate que les règles utilisent des marqueurs différents selon l'opération d'édition. Ceci permet à plusieurs opérations de s'appliquer au même endroit, pour autant que le contexte s'y prête.

```
[RULE]
  <INS> → b / 2
  b <SUB> → [fghvn] / 1
  b <DEL> → <ISDEL> : : [fghvn] _ / 2
  b <DEL> → <ISDEL> : : _ [fghvn] / 2
  b <DEL> → <ISDEL> / 3
  ...
  <ISDEL> → \x00
```

FIG. 15.25: Fichier Ovide : opérations d'édition

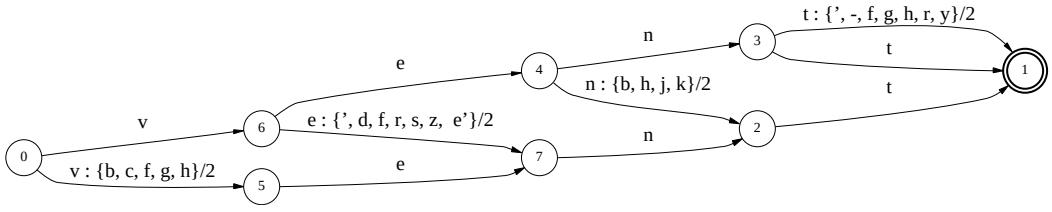


FIG. 15.26: Distance d'édition : substitution filtrée

On constate en outre que les règles ne sont pas optionnelles, mais obligatoires. Ceci est dû à la présence du marqueur : la règle ne s'applique que si le marqueur est présent, mais *doit* s'appliquer dans ce cas.

Dans notre exemple, la règle qui gère la suppression (marqueur `DEL`) ne supprime pas le symbole, mais le projette sur un autre marqueur, `ISDEL`. Ce marqueur note la suppression et joue le rôle d'un *bloqueur* : il empêche que des caractères qui n'étaient pas contigus dans le mot le deviennent du fait de cette suppression. D'autres règles, dans ce cas, pourraient en effet s'appliquer à tort. Lorsque toutes les règles ont été appliquées, le bloqueur est enfin supprimé, ce qui active la règle de suppression, sans risque d'erreur de modélisation.

La Figure 15.26 montre le résultat obtenu sur le mot *vent*. Pour des raisons de lisibilité, la seule opération proposée est la substitution. On constate que les caractères de substitution font toujours partie du contexte direct du caractère substitué. Le FST *Edit*, qui gère environ 200 règles d'édition contextualisées, prend 2,9 Mo.

Le clavier. Le filtre typographique est intrinsèquement lié à la disposition des touches sur le clavier. Or, nous avons voulu rendre le processus de génération du filtre le plus indépendant possible d'un clavier donné. Pour ce faire, nous proposons d'employer un modèle de clavier, décrit au format texte. Un ensemble de scripts permettent ainsi de générer rapidement le fichier Ovide correspondant à la machine *Edit*.

	é		’		-	è		ç	à	
a	z	e	r	t	y	u	i	o	p	^
q	s	d	f	g	h	j	k	l	m	ù
	w	x	c	v	b	n				

TAB. 15.1: Modèle de clavier (valeurs accessibles sans touche de fonction)

Certaines touches d'un clavier classique présentent plusieurs valeurs, dont l'une est accessible directement, tandis que les autres sont accessibles *via* les touches de fonction (*Shift*, *Ctrl*, *Alt* et *Alt Gr*). Notre modèle ne tient compte que des valeurs directement accessibles qui peuvent apparaître dans des tokens lexicaux (cf. Table 15.1).

Or, les majuscules ne font pas partie des valeurs directement accessibles. Pour combler ce manque, notre filtre typographique est complété par la recherche relâchée. Une recherche typographique d'un OOV dans le dictionnaire correspond donc à la cascade de compositions suivante :

$$OOV \circ (Match \circ Long \circ Edit) \circ Lexicon$$

Originalité du modèle. Contrairement aux approches basées sur des machines à états finis qui ont été présentées dans l'état de l'art (cf. Section 14.4.1.4), le modèle que nous proposons ne modifie pas le mécanisme classique de la composition. Notre approche autorise une composition classique, parce que les contraintes sur l'intersection acceptée sont contenues dans les machines elles-mêmes. Cette nouvelle approche offre les avantages suivants :

1. Une modélisation simplifiée : il n'est pas nécessaire d'appréhender, dans un seul et même effort de réflexion, les opérations permises et le nombre ou le lieu de ces opérations.
2. Une flexibilité de la limite imposée : contrairement aux parcours augmentés proposés dans l'état de l'art, notre approche permet de poser des contraintes différentes sur les différentes opérations d'édition : il est facile de décrire les combinaisons d'opérations que l'on privilégie ou que l'on rejette, ou de jouer sur les distances entre les opérations. Il a par exemple été très facile d'interdire la succession d'une suppression et d'une insertion.
3. Les marqueurs ajoutent à l'originalité de la méthode : ce sont des repères sans ambiguïté, qui évitent le déploiement d'expressions régulières complexes.
4. La conservation de la distance d'édition dans le poids du candidat permet de différencier les candidats en fonction du nombre d'opérations d'édition qu'ils ont nécessité.

15.6.1.3 Le filtre phonétique

Positionnement du problème. L'objectif de la distance phonétique est de déterminer les mots du lexique qui se prononcent de manière équivalente à l'OOV. Un OOV comme *plène*, par exemple, pourrait dans ce cas se voir attribuer des candidats comme *pleine* ou *plaine*.

Un système de phonétisation, nous l'avons expliqué dans la description du principe de la synthèse (cf. Section 12.1), prend normalement en compte la catégorie d'un mot pour le phonétiser. C'est ainsi qu'il peut par exemple distinguer « *couvent*, `NOUN` » de « *couvent*, `VERB` ».

Or, dans le cas d'un OOV, le système ne connaît pas la catégorie du mot. Il est dès lors nécessaire de générer l'ensemble des phonétisations *possibles*. Chaque phonétisation produira des candidats, parmi lesquels on espère voir figurer la solution.

Source de la phonétisation. Nous avons choisi d'utiliser le lexique de phonétisation d'eLite comme base de travail. Dans ce lexique qui compte 282 411 formes, chaque mot est accompagné de sa catégorie et de sa phonétisation :

```
abaissant VERB a b E _ s _ a~ _ _
...
couvent NOUN k u _ v a~ _ _
couvent VERB k u _ v _ _ _
...
évanouissait VERB e v a n u _ i s _ E _ _
...
```

Dans les exemples ci-dessus, les *underscores* représentent le silence. Ils permettent d'aligner chaque phonème sur la première lettre du graphème auquel il correspond.

Note 15.6.2. Dans ce document, nous employons le terme *graphème* dans le sens de *suite de caractères qui correspondent à un seul phonème*. Le graphème se prononce donc comme un tout.

Grâce aux *underscores*, il est donc aisé de segmenter un mot en la suite de graphèmes qu'il contient. Par exemple, *évanouissait* sera segmenté :

`é | v | a | n | ou | i | ss | ait`

La catégorie qui accompagne chaque terme dans le lexique ne nous intéresse pas, étant donné que la catégorie de l'OOV est indéfinie.

Principe. Nous désirons déterminer, à partir du lexique, l'ensemble des graphèmes qui se prononcent de manière identique, afin de construire des règles du type :

$$\texttt{a} \overset{?}{\rightarrow} \texttt{(b|c|d) / } w$$

qui exprime que le graphème `a` *peut* être réécrit par les graphèmes phonétiquement équivalents `b`, `c` ou `d`, et reçoit le poids w.

Contrainte. Nous faisons l'hypothèse qu'une modification graphique ne peut être réalisée que si le *contexte* le permet. Ceci signifie que la graphème proposé en réécriture doit avoir été rencontré, parmi les mots du lexique, dans le contexte où l'on désire l'appliquer. Selon cette hypothèse, les règles prennent la forme suivante :

$$\texttt{a?}\rightarrow \texttt{b :: e \_ f /}\ w_1$$
$$\texttt{a?}\rightarrow \texttt{(c|d) :: g \_ h /}\ w_2$$

Nous faisons l'hypothèse qu'un contexte d'un seul caractère est suffisant. Bien sûr, le contexte peut également être le début et/ou la fin du mot. Par exemple, « *ent* » peut être réécrit « *ant* » en finale derrière v, parce que la forme *devant* a été rencontrée dans le lexique.

$$\texttt{ent?}\rightarrow \texttt{ant :: v \_ \$ /}\ 1.5$$

Construction des règles. L'algorithme de construction des règles de réécriture est présenté en Pseudocode 35. Le fichier Ovide contient un peu plus de 18 000 règles. Le FST correspondant représente quant à lui 7,3 Mo.

Application du modèle. Lorsque nous appliquons le filtre phonétique, le nombre de candidats varie énormément d'une forme à l'autre, comme l'illustre la Figure 15.27. Le mot *ovant*, par exemple, présente deux erreurs phonétiques et génère deux candidats. Le mot *plène*, par contre, qui ne présente qu'une erreur phonétique, génère une vingtaine de candidats. Parmi ceux-ci figurent les candidats *pleine* et *plaine*, qui apparaissent comme les candidats les plus probables.

Originalité du modèle. Notre approche se distingue de la distance phonétique proposée dans l'état de l'art (cf. Section 14.4.1.3), où les transformations phonétiques sont réalisées en cours de traitement. Dans notre modèle, la conversion *graphèmes* → *phonèmes* n'est réalisée qu'à l'entraînement, de manière à construire des ensembles de graphèmes phonétiquement équivalents. En cours de traitement, le FSM convertit *directement* un graphème en une liste de graphèmes apparus dans le même contexte. La méthode permet donc de supprimer une étape coûteuse, sans modification du principe en lui-même.

En outre, notre distance phonétique est contextualisée : un graphème phonétiquement équivalent n'est proposé que s'il a été rencontré dans le contexte adéquat, réduit à un caractère de part et d'autre.

1 : **Pour chaque** forme du lexique,
2 : **Pour chaque** graphème de la forme,
3 : Mémoriser le phonème et le contexte d'apparition.
4 : **Pour chaque** graphème Gra,
5 : **Pour chaque** phonème Pho auquel Gra correspond,
6 : **Pour chaque** contexte $Ctxt$ dans lequel Gra est prononcé Pho,
7 : Créer une règle qui permet à tout graphème prononcé Pho d'être réécrit Gra dans le contexte $Ctxt$.

Pseudocode 35: Construction des règles de distance phonétique

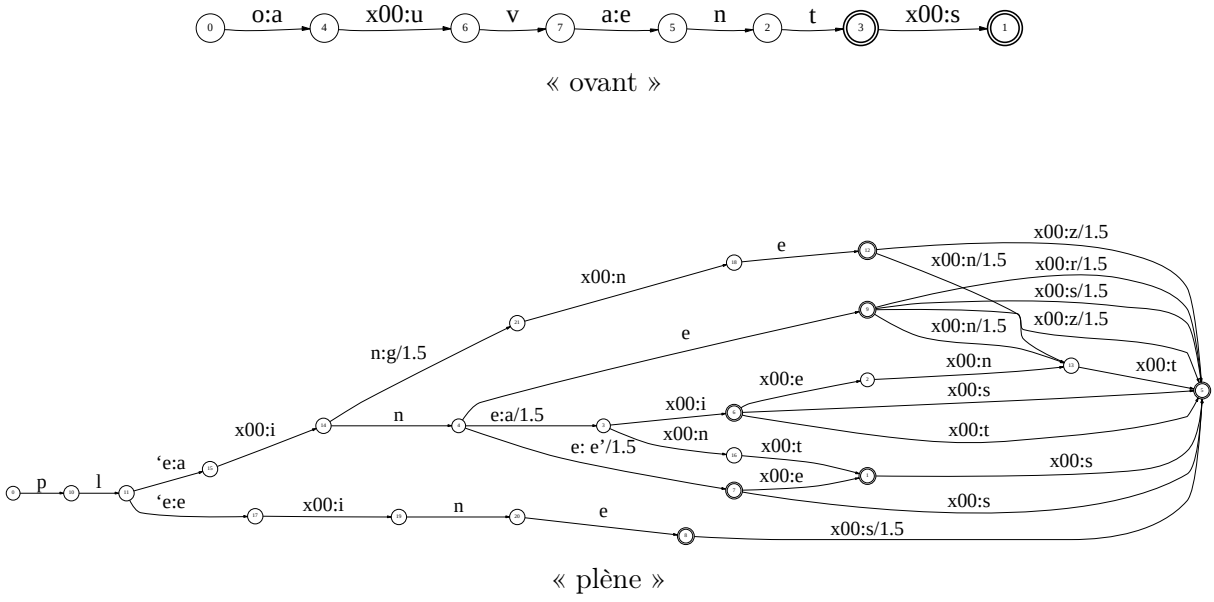


FIG. 15.27: Illustration du filtre phonétique. Afin d'en faciliter la lecture, les machines présentées ont été allégées des analyses morphologiques

15.6.2 Correction flexionnelle

La modélisation de l'ensemble de la correction flexionnelle repose sur l'hypothèse suivante :

Hypothèse 15.6.2 (Modèle de langue flexionnel)**.** *Les performances des modèles de langue en analyse syntaxique laissent penser qu'un modèle de langue, combinant les catégories syntaxiques et les traits grammaticaux, peut efficacement gérer les erreurs d'accord présentes dans une phrase.*

La Figure 15.28 présente le schéma-bloc de la correction flexionnelle, dont l'objectif est de corriger les erreurs d'accord dans une phrase dont l'analyse syntaxique a été réalisée. Le FSM sur lequel travaille l'analyse flexionnelle correspond donc à la totalité de la phrase, mais ne contient que le chemin le plus probable.

Seuls les tokens lexicaux de la phrase doivent être traités par l'analyse flexionnelle : les autres tokens ont été détectés parce qu'ils respectent un format strict, et ne présentent donc pas d'erreur d'accord. De ce fait, un test dichotomique segmente le FSM dans un vecteur de FSMs, dont chaque position est soit une suite de tokens lexicaux, soit une suite d'autres tokens.

Chaque FSM correspondant à une suite de tokens lexicaux est traité séparément, en quatre étapes. Une première étape tâche de détecter les formes qui présentent une erreur d'accord potentielle. Pour chaque forme détectée, le système ajoute dans la machine *l'analyse* qu'il propose en remplacement.

Une seconde étape complète le FSM en y insérant les flexions correspondant aux analyses ajoutées par la première étape. Il est important de mentionner que ces flexions corrigées sont *ajoutées* : elles ne remplacent pas les flexions originales, mais constituent des *alternatives*.

La troisième étape est une évaluation statistique, qui détermine la pertinence des différentes flexions possibles pour chaque forme de la phrase.

La quatrième étape est le calcul du meilleur chemin, qui ne retient les flexions insérées précédemment que si elles appartiennent au chemin le plus probable.

Lorsque le vecteur complet a été traité, un seul FSM est reconstitué et est retourné à l'algorithme principal. Ce FSM est prêt à être sauvegardé dans la DLS.

Si l'on excepte la recherche du meilleur chemin, le système de correction flexionnelle en lui-même se répartit donc entre 3 FSMs distincts : la détection flexionnelle $DetFlex$, la génération flexionnelle $GenFlex$ et le modèle de langue $NFlex$, appliqués sur la phrase S par composition :

$$S \circ (DetFlex \circ GenFlex \circ NFlex)$$

La partie délicate de ce système est la construction des machines de détection et de génération. Dans un premier point, nous en décrivons les principes, avant d'en détailler

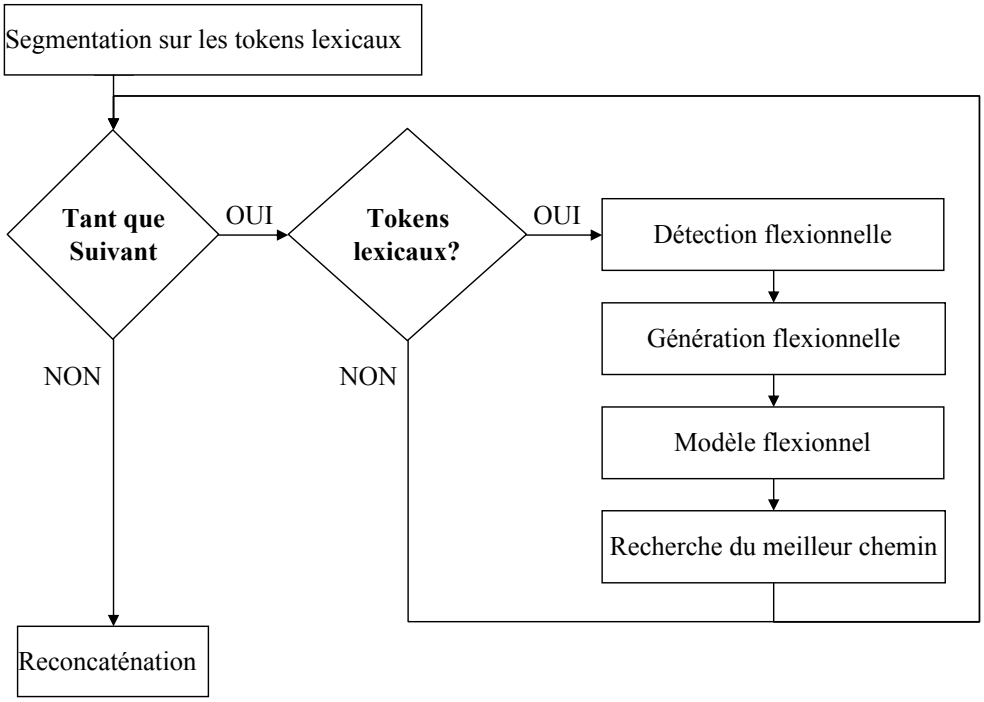


FIG. 15.28: Schéma-bloc de la correction flexionnelle

la construction dans un second point. Un troisième point décrit ensuite les réflexions qui ont conduit au modèle de langue flexionnel que nous utilisons.

Note 15.6.3. Dans ces différents points, nous parlons de *catégorie fléchie*, c'est-à-dire une catégorie accompagnée d'une désinence (genre, nombre, personne).

15.6.2.1 Principes de la détection et de la génération

La détection. Nous proposons une approche par règles, qui consiste à rechercher des divergences flexionnelles au sein d'une fenêtre d'analyse. Lorsqu'une divergence est détectée, le système *propose* une analyse flexionnelle qui s'*ajoute* à l'analyse flexionnelle initiale.

La fenêtre d'analyse couvre au moins 2 catégories fléchies. Une règle concerne toujours une seule catégorie fléchie de la fenêtre, et peut traiter soit un seul trait, soit plusieurs traits simultanément. Voici quelques exemples de règles, concernant le trait *nombre* dans une fenêtre de taille 3 :

1. Le trait cible vaut Sg et les traits contextuels valent Pl. Le trait cible devrait probablement valoir Pl. Ce cas est idéal, et doit être privilégié par le système :

```
Sg ?→ Pl :: Pl _ Pl / 1
Sg ?→ Pl :: _ Pl Pl / 1
...
```

2. Le trait cible vaut Sg, un trait contextuel vaut Pl tandis que l'autre est neutre. Il se peut donc que le trait cible doive prendre la valeur Pl. La décision est moins évidente :
   ```
   Sg?→ Pl :: NND _ Pl / 1.5
   Sg?→ Pl :: _ Pl NND / 1.5
   ...
   ```
3. Le trait cible et un trait contextuel valent Sg, tandis que l'autre trait contextuel vaut Pl. La nécessité de changer la valeur du trait cible est peu probable :
   ```
   Sg?→ Pl :: Sg _ Pl / 2
   Sg?→ Pl :: _ Pl Sg / 2
   ...
   ```

Les règles que nous générons concernent principalement les configurations suivantes :

1. L'accord du sujet et du verbe, que le sujet soit un pronom ou un nom.
2. L'accord du nom et du verbe séparés par le pronom relatif *qui*.
3. L'accord du verbe *être* et de son prédicat.
4. L'accord du déterminant avec le nom ou l'adjectif.
5. L'accord du nom avec l'adjectif ou le participe passé.
6. L'accord de l'adjectif et du nom.
7. La nécessité d'un infinitif après un verbe conjugué autre qu'un auxiliaire. Par exemple, *il doit manger*.
8. La nécessité d'un participe après un auxiliaire. Par exemple, *il a mangé*.

Ces configurations sont traitées automatiquement, à partir d'un script qui génère l'ensemble des variations possibles en termes de genre, de nombre et de personne. En tout, ces quelques configurations correspondent à environ 500 règles.

Le script accepte que des éléments s'intercalent entre les catégories d'une configuration. Par exemple, la configuration 1 accepte entre le sujet et le verbe un seul adverbe et un ou deux pronoms compléments. La configuration chargée par le script a de ce fait la forme suivante :

```
(PRONSJ | NOUN) "ADV? PRONPCP{,2}" VERB
```

où :

- `PRONSJ` fait référence à l'ensemble des pronoms qui peuvent être sujets.
- `PRONPERCP` fait référence à l'ensemble des pronoms qui peuvent être compléments. La syntaxe `{,2}` indique que le pronom peut être présent 0, 1 ou 2 fois.
- Les guillemets indiquent au script que les éléments qu'il contient font partie de la règle, mais que la règle ne s'y applique pas.

La génération. A ce stade, le système de détection a inséré dans la machine, s'il y a lieu, des variations flexionnelles sous la forme de nouveaux traits grammaticaux.

Par exemple, la forme *beau*, dans « *les beau_ enfants* », est maintenant associée à deux analyses : une analyse originale, `Msc,Sg,PND`, et une nouvelle analyse `Msc,Pl,PND`.

L'objectif de l'étape de génération est de modifier la flexion lorsqu'elle est suivie de la nouvelle analyse. Dans l'exemple présenté, il faut faire correspondre l'analyse `Msc,Pl,PND` à *beaux*...

Or, la transformation à appliquer n'est pas constante et ne dépend pas exclusivement de la désinence :

1. La flexion à générer peut impliquer un ajout de caractères par rapport à la flexion existante (*beau* → *beaux*), mais également une suppression (*beaux* → *beau*) ou un remplacement (*beau* → *belle*).
2. La flexion correspondant à une désinence particulière varie d'une classe flexionnelle à l'autre : *beaux*, *lourds*, ...

Il est donc nécessaire d'en tenir compte dans l'algorithme de construction des règles de génération. Cet algorithme, présenté en Pseudocode 36, se fonde sur le lexique de flexions d'eLite présenté en Section 13.4.1.2. Son objectif est de construire un nombre minimum de règles, de manière à compacter au mieux la taille de la machine, tout en autorisant toutes les transformations flexionnelles possibles. Pour ce faire, le principe est de réduire chaque transformation possible $f_1 \rightarrow f_2$ aux seuls caractères divergents, précédés d'un caractère commun. Le but est de rassembler en une seule règle les règles proches. Les règles créées sont ensuite imprimées, dans un ordre décroissant, depuis celle qui présente la terminaison f_1 la plus longue. De la sorte, une règle générale ne peut être appliquée à la place de la règle spécifique au cas à traiter.

Sans suppression des préfixes communs, l'algorithme génère un peu plus de 200 000 règles. Grâce à la suppression, seules 3 240 règles sont conservées, ce qui permet au FSM correspondant de tenir dans 2,4 Mo.

1 : **Pour chaque** classe de flexion F_i,
2 : **Pour chaque** flexion f_1 de F_i,
3 : **Pour chaque** flexion f_2 de F_i et sa désinence d_2,
4 : Supprimer le préfixe commun à f_1 et f_2, sauf la dernière lettre du préfixe commun.
5 : **Si** la règle n'existe pas dans la liste L,
6 : Créer dans L la règle $f_1 \rightarrow f_2 : : \_\ d_2$, et lui associer F_i.
7 : **Sinon**
8 : Ajouter F_i dans la liste des classes auxquelles la règle s'applique.
9 : **Pour chaque** règle $f_1 \rightarrow f_2$ de L, dans l'ordre inverse de la longueur de f_1,
10 : Imprimer $f_1 \rightarrow f_2 : : \_\ (F_i|\ldots|F_j)\ d_2$

Pseudocode 36: Construction des règles de génération flexionnelle

15.6.2.2 Construction de la détection et de la génération

La section précédente a décrit les principes qui gouvernent le système de détection et de génération. Les règles ne peuvent cependant être écrites telles qu'elles ont été présentées. En effet, les modèles doivent tenir compte des contraintes suivantes :

1. Un trait grammatical qui a été modifié par une règle ne peut être à son tour la cible d'une autre règle. Imaginons une entrée $BABC$ et les deux règles
   `A ?→ B :: B _ B / 1`
   `B ?→ C :: _ B C / 2`
   L'application successive de ces règles va transformer $BABC$ en $BCBC$, alors que le résultat voulu est $BBBC$. Il est donc nécessaire d'empêcher l'application de règles en cascade. Un marqueur de type *masqueur* est tout désigné dans ce cas.
2. Une désinence additionnelle ne se distingue en rien d'une désinence originale. Or, les règles de génération ne peuvent s'appliquer que dans le cas de désinences additionnelles. Un marqueur de type *déclencheur* est donc nécessaire.

Ces constats nous ont conduit à redéfinir les règles de réécriture des deux fichiers. Les Figures 15.29 et 15.30 illustrent les modifications apportées.

La détection. Les règles illustrées en Figure 15.29 modélisent quelques cas de transformation du trait grammatical « nombre », de *singulier* en *pluriel*. Ces règles de détection appliquent le masqueur défini, et non la valeur désirée. Une règle insère ensuite le déclencheur, et se base pour cela sur la présence du masqueur. Les règles qui ferment le fichier activent les conversions désirées, sans risque de réécritures en cascade. Après l'application des règles sur le FSM représentant la phrase, les désinences suspectes sont complétées de nouveaux traits grammaticaux, systématiquement suivis du déclencheur.

La Figure 15.31 illustre le résultat de l'application des règles de détection sur la phrase « les beau_ enfants ». Seule la partie correspondant à « beau » est représentée.

La génération. Les règles de la Figure 15.30 présentent toutes le déclencheur, qui indique que la génération de la flexion adéquate peut être réalisée. La règle projette la terminaison de la flexion erronée *et* le déclencheur sur la flexion correcte. Le déclencheur est donc supprimé dès l'application de la règle désirée, de sorte qu'aucune autre règle ne peut plus s'appliquer.

La Figure 15.32 montre le résultat de l'application des règles de génération sur l'exemple de la Figure 15.31.

15.6.2.3 Le modèle de langue flexionnel

Afin de gérer les erreurs flexionnelles, notre intention initiale était d'augmenter le modèle syntaxique $P(T)$ de manière à ce qu'il intègre l'estimation des flexions. Ceci consiste *a priori* à évaluer la probabilité d'un tri-gramme de catégories fléchies, où chaque catégorie

```
[CLASSIN]
  NUMBER    [Pl Sg NND]
  DECL      &1
  SGMASK    &2
  PLMASK    &3
[RULE]
  # les detections
  Sg?→ <PLMASK> :: Pl [^<NUMBER>]* _ [^<NUMBER>]* Pl / 1
  Sg?→ <PLMASK> :: NND [^<NUMBER>]* _ [^<NUMBER>]* Pl / 1.5
  Sg?→ <PLMASK> :: Pl [^<NUMBER>]* _ [^<NUMBER>]* Sg / 2
  # insertion du declencheur
  \x00 → <DECL> :: _ . . [<SGMASK><PLMASK>]
  # conversion finale des masqueurs
  <SGMASK> → Sg
  <PLMASK> → Pl
```

FIG. 15.29: Fichier Ovide : détection des erreurs d'accord

```
[RULE]
  elle <DECL> → eau :: _ "mf,sp,-x+,-elle+eau" Msc Sg PND
  ...
  u <DECL> → ux :: _ "mf,sp,-x+,-elle+eau" Msc Pl PND
  ...
  e <DECL> → ent :: _ (26 | 6 | 84) GND Pl 3rd
  e <DECL> → ées :: _ 6 Fem Pl PND PARTPASSE
  e <DECL> → és :: _ 6 Msc Pl PND PARTPASSE
  e <DECL> → \x00 :: _ (26 GND Sg 3rd) | ("mf,sp,-s+,-e+" Msc Sg PND))
```

FIG. 15.30: Fichier Ovide : génération des flexions

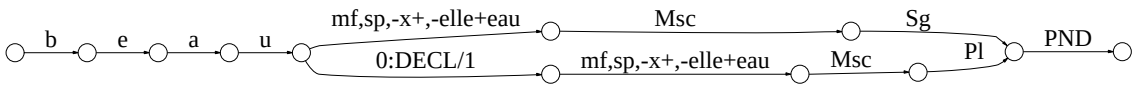


FIG. 15.31: Erreurs d'accord : détection. Seule la partie de la machine modifiée est illustrée

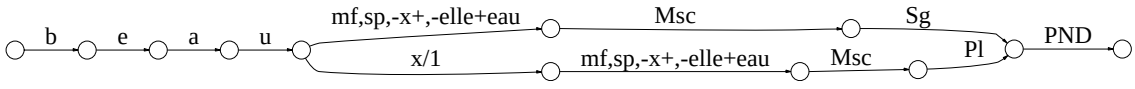


FIG. 15.32: Erreurs d'accord : génération sur l'exemple de la Figure 15.31

t_i est accompagnée de sa désinence d_i :

$$p(T^D) \approx p(t_i^{d_i} | t_{i-2}^{d_{i-2}}, t_{i-1}^{d_{i-1}}) \tag{15.6.2.1}$$

En faisant varier les valeurs des 3 traits grammaticaux que nous utilisons dans le système (genre, nombre et personne), nous pouvons construire 36 désinences différentes (3*3*4). Etant donné que notre analyseur utilise 60 catégories, l'ensemble des catégories fléchies compte 2160 combinaisons possibles. De ce fait, un modèle syntaxique fondé sur l'ensemble des catégories fléchies devrait évaluer 10 077 696 000 tri-grammes différents. En d'autres termes, un tel modèle n'est pas envisageable...

D'un autre côté, l'estimation des désinences de manière complètement indépendante des catégories n'aurait probablement aucune signification véritable.

Sur la base de ces réflexions, nous proposons d'estimer un modèle de langue flexionnel, construit sur l'ensemble des désinences et un nombre réduit de catégories, qui jouent le rôle de *pivots syntaxiques*. Le choix des catégories-pivots se base sur les constats suivants :

1. De nombreuses catégories ne rassemblent que des mots invariables, qui servent principalement à l'articulation de la phrase. Il s'agit par exemple des conjonctions de coordination ou de subordination, des prépositions ou des adverbes. Nous les rassemblons de ce fait dans une catégorie « mots-outils » ou « invariables ».
2. D'une part, le verbe occupe une place particulière dans la phrase, et a un comportement syntaxique distinct de celui du nom. D'autre part, de nombreuses catégories modifient ou complètent le nom. Une distinction utile peut dès lors être conservée entre les catégories verbales du système (auxiliaires, participes, etc.) et les catégories non verbales (noms, adjectifs, déterminants, etc.).

Sur cette base, nous avons conservé 3 catégories-pivots :

1. Les mots-outils (`INVARIABLE`).
2. Les verbes (`VERB`).
3. Les non-verbes (`NOUN`).

Le modèle flexionnel se définit donc tel que présenté en Equation 15.6.2.1, si ce n'est que les catégories fléchies sont construites à partir des catégories-pivots uniquement. En outre, la catégorie *mot-outil*, qui est invariable, peut uniquement être accompagnée de la désinence neutre (`GND`,`NND`,`PND`).

Ces contraintes permettent de réduire le nombre de catégories du modèle à 73 ($2 * 36 + 1$), ce qui autorise 389 017 tri-grammes différents. Le modèle est maintenant estimable.

Le FSM correspondant à ce modèle a été construit selon la même méthode que le modèle syntaxique (cf. Section 15.5.1.2) : nous avons généré un fichier Ovide, dans lequel des règles pondèrent l'ensemble des n-grammes nécessaires : bi-grammes initiaux, tri-grammes initiaux et tri-grammes. Comme le modèle syntaxique, le modèle flexionnel est lissé par interpolation linéaire. La machine compilée représente 8 Mo.

15.6.2.4 Originalité du modèle

On aura constaté la similarité avec l'approche linguistique basée sur la relaxation des contraintes syntaxiques (cf. Section 14.5.1). Cependant, il fallait un échec de l'analyse pour que le système linguistique relâche les contraintes.

Notre méthode génère une flexion supplémentaire *dès* qu'un tri-gramme présente une configuration qui *pourrait* être erronée. La flexion supplémentaire n'est qu'une possibilité de plus, dans un système pondéré qui déterminera automatiquement la flexion qui lui paraît la plus probable. La flexion ajoutée n'est donc pas l'unique solution possible du fait d'un échec antérieur. En outre, cette solution additionnelle est désavantagée par rapport à la solution originale, étant donné qu'une distance la pondère.

15.6.3 Les points-clefs des modèles de correction

La correction des OOVs est réalisée sous la forme d'une distance d'édition complexe, qui modélise soit une distance typographique, soit une distance phonétique. Dans les deux cas, la distance est représentée sous la forme d'un filtre de composition, capable de limiter le nombre des solutions sur la base de contraintes variables, parce que dépendantes des caractéristiques de la forme à corriger. La distance typographique tient compte de la longueur de la forme et de la disposition du clavier, tandis que la distance phonétique tient compte du contexte graphémique pour autoriser les transformations. Les deux modèles tiennent compte du nombre d'opérations qui ont été nécessaires pour trouver un candidat donné.

La correction flexionnelle se base sur un certain nombre de règles pour détecter une éventuelle erreur. Lorsqu'une erreur potentielle est détectée, le modèle insère une ou plusieurs *alternatives pondérées*, parmi lesquelles un modèle de langue, défini sur quelques catégories-pivots et l'ensemble des désinences, et choisit la meilleure solution.

15.7 Evaluation

Les objectifs globaux de l'évaluation réalisée sont les suivants :

1. Evaluer l'analyse morpho-syntaxique dans son ensemble, en termes de taux d'étiquetage syntaxique et de temps de traitement.
2. Evaluer la pertinence des modèles de correction *en contexte*. Le but est de distinguer les cas que le système gère de ceux qu'il ne gère pas, et de déterminer dans quelle mesure le système corrompt le texte en y introduisant des erreurs qui n'y figuraient pas.

Tous les tests ont été réalisés sous Windows XP, sur un PC portable Dell Inspiron 8600 possédant un processeur Intel Pentium Mobile cadencé à 1,7 GHz et pourvu de 1,0 Go de RAM.

Avant d'aborder l'évaluation proprement dite, nous faisons quelques remarques préliminaires concernant les modèles employés et l'évolution des données entre les deux versions du système d'analyse linguistique.

15.7.1 Remarques préliminaires

15.7.1.1 Les modèles employés et leurs combinaisons

Le modèle de base. De tous les modèles présentés dans ce chapitre, le seul qui soit toujours inclus dans l'analyse est le modèle syntaxique $P(T)$, qui permet de choisir le meilleur chemin dans le treillis de solutions construit par l'ensemble du processus. Lorsque l'analyse se limite à ce modèle, nous parlons de *modèle de base*. Le modèle de base correspond à l'ancienne analyse syntaxique *sans modèle lexical*.

Le corpus utilisé pour l'entraînement de ce modèle est le même que celui employé lors de l'évaluation de la première analyse syntaxique (cf. Section 13.5.3.5) : « Le mot et l'idée », constitué par l'AUPELF-UREF et comprenant 66 619 mots.

Au cours de cette première évaluation, nous avions constaté que les différentes méthodes de lissage implémentées ne produisaient pas de résultats significativement différents. Nous en avions conclu que le corpus d'entraînement était insuffisant pour assurer une estimation correcte des différentes méthodes de lissage. Dans le cadre de cette nouvelle évaluation, nous n'avons de ce fait retenu qu'une seule méthode : *Kneser-Ney*, dont les performances ne sont pas les plus élevées, mais qui désambiguïse mieux les homographes hétérophones.

Les modèles facultatifs. Outre le modèle $P(T)$, le nouvel analyseur morpho-syntaxique peut combiner un ou plusieurs des modèles suivants :

1. Le modèle lexical des IVs.
2. Le modèle lexical des OOVs.
3. Le modèle de recherche relâchée.
4. Le modèle de casse (noms propres et acronymes).
5. Le modèle syntagmatique.
6. Le modèle de distance typographique (correction des OOVs).
7. Le modèle de distance phonétique (correction des OOVs).
8. Le modèle flexionnel pour la correction des erreurs d'accord (correction de l'ensemble des formes lexicales).

La déclaration des modèles utilisés est réalisée dans un fichier d'initialisation, pour plus de flexibilité.

Nous rappelons que le modèle lexical des IVs est inclus dans la machine qui représente le lexique, tandis que le modèle lexical des OOVs est inclus dans la machine qui réalise l'analyse morphologique des OOVs. Or, ces machines sont *nécessaires* au processus, qui doit

au moins pouvoir attribuer les analyses morphologiques aux formes lexicales de la phrase. La suppression de ces modèles est donc obtenue, au chargement, par la suppression de la pondération des machines concernées [11].

Les six autres modèles, par contre, sont chacun isolés dans une machine dédiée. Lorsque l'un de ces modèles n'est pas nécessaire, la machine correspondante n'est donc pas chargée.

Note 15.7.1 (Référence aux modèles employés)**.** Dans le cadre de cette évaluation, nous identifions une combinaison de modèles à l'aide d'une séquence de huit valeurs. Dans cette séquence, toute position i correspond au modèle i de la liste présentée ci-dessus. La position vaut i si le modèle est utilisé, 0 sinon. Par exemple, le modèle de base, qui n'emploie aucun des modèles facultatifs, vaut 00000000, tandis que la combinaison du modèle typographique et du modèle flexionnel correspond à la séquence 00000608.

15.7.1.2 Evolution des données

Le modèle de base (00000000) correspond à l'ancienne analyse syntaxique dépourvue de modèle lexical, tandis que le modèle 10000000 correspond à l'ancienne analyse syntaxique incluant le modèle lexical ; en effet, seul le modèle lexical des IVs existait dans la première version de l'analyse syntaxique.

Si on limite le nouveau système à ces configurations, les deux systèmes sont bien entendu équivalents. Or, comme nous aurons l'occasion de le constater en analysant les tests concernés, le nouveau système, dans ces configurations, obtient environ 1% de mieux que l'ancienne analyse syntaxique. Ceci est dû à une évolution des données entre les deux évaluations :

1. Nous avons remarqué et corrigé un manque de cohérence dans l'étiquetage du corpus d'entraînement. En effet, certains cas identiques étaient étiquetés différemment selon la position dans le corpus, ce qui induisait le système en erreur.
2. Les lexiques de lemmes et de flexions ont été corrigés et complétés. Il faut cependant noter que les lexiques n'ont pas été adaptés au corpus d'entraînement qui, lors de l'évaluation, contenait toujours 1 201 mots hors vocabulaire. Les seules modifications apportées au lexique étaient d'ordre linguistique : rectifications de flexions erronées, modifications des catégories attribuées, etc.

15.7.2 Evaluation de l'analyse morpho-syntaxique

L'objectif est d'estimer le taux d'étiquetage et la vitesse de traitement. Pour ce faire, nous prenons les points de vue suivants :

1. L'évaluation séparée des différents modèles présentés.

[11] La suppression de la pondération est réalisée à l'aide de la méthode `FSM_Balance`, définie dans notre bibliothèque, qui permet d'appliquer un facteur aux poids d'un FSM. Si le facteur vaut 0, les poids des transitions sont annulés.

2. L'analyse des différentes combinaisons possibles des modèles dédiés à l'analyse morpho-syntaxique.
3. L'impact des modèles de correction sur la qualité de l'étiquetage.

Comme lors de l'évaluation du premier système, ces différents tests ont été réalisés par 10-fold-cross-validation : le corpus a été segmenté en 10 parties égales, chaque partie servant 9 fois à l'entraînement et 1 fois de corpus d'évaluation.

15.7.2.1 Les modèles isolés

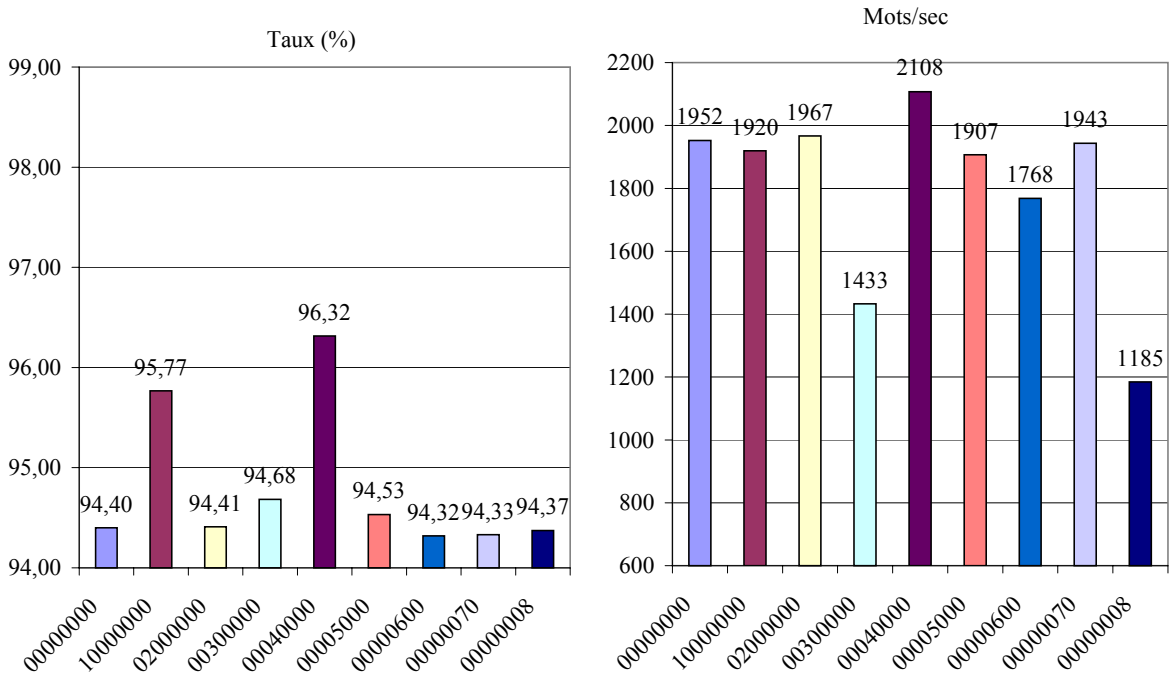
La Table 15.2 confronte les résultats du modèle de base aux résultats obtenus par l'adjonction d'un seul modèle facultatif.

Le modèle de base et le modèle lexical des IVs (10000000) produisent des résultats satisfaisants. Comme nous l'avons signalé, l'évolution des données entre les deux évaluations explique l'amélioration des résultats d'environ 1% entre l'ancien et le nouveau système, puisque l'on passe de 93,66 à 94,41% sans modèle lexical, et de 94,63 à 95,77% avec modèle lexical. Le gain est évidemment appréciable, puisqu'il représente environ 600 mots de mieux sur l'ensemble du corpus. L'impact réel sur le système de synthèse est cependant réduit, parce que la différence principale se situe au niveau de l'attribution des catégories *nom* et *adjectif*... Cette amélioration montre donc principalement une plus grande cohérence du corpus d'entraînement.

Le meilleur modèle est incontestablement le modèle de casse (00040000), dont les résultats sont 2% supérieurs à ceux du modèle de base. Ceci est dû au fait que le corpus contient 1 777 noms propres et 237 acronymes. La décision *a priori* d'attribuer ces catégories en fonction de la casse est donc fructueuse, en tout cas dans le contexte de ce corpus principalement journalistique. On constate en outre que la politique de rejeter toute autre analyse lorsque le mot est OOV profite au temps de traitement, qui est de loin le meilleur obtenu. Ces tendances, obtenues sur le corpus d'entraînement, demandent une confirmation sur d'autres corpus (cf. *Infra*).

Le modèle lexical des OOVs (02000000) et la recherche relâchée (00300000) n'améliorent pas les résultats de manière significative. Ceci n'est pas étonnant, sur un corpus qui contient peu d'erreurs de casse ou d'accentuation, comme nous aurons l'occasion de le détailler dans l'évaluation de la correction des OOVs. On constate en outre que la recherche relâchée pénalise fortement le temps de traitement, parce qu'elle ajoute de nombreuses transitions au treillis de solutions. le modèle semble donc inutilement lourd. Il a en fait été pensé pour être combiné à d'autres modèles, comme nous le verrons par la suite.

Le modèle syntagmatique (00005000) produit un étiquetage légèrement meilleur que le modèle de base. Nous l'avons expliqué, ce modèle a été prévu pour gérer certaines structures simples, dont l'analyse dépasse les possibilités de la fenêtre du n-gramme. C'est le cas, par exemple, de la négation *plus* dans la phrase « *je n'en peux plus* ». Tous les exemples de ce type, présents dans le corpus, sont corrigés par ce modèle, ce qui est un plus indéniable



TAB. 15.2: Performances par modèle

pour la qualité de la parole générée. La faible amélioration apparente, en terme de taux, est due au fait que le corpus contient peu de structures de ce type.

Les modèles de correction induisent systématiquement une diminution des résultats. Nous le verrons dans l'évaluation concernée, le corpus contient peu d'erreurs, alors que 1 500 mots n'appartiennent pas à notre lexique. Les « fausses alarmes » sont donc à l'origine de cette dégradation des résultats. On constate en outre que la correction flexionnelle (00000008) est le modèle qui, utilisé *isolément*, ralentit le plus le processus global. Ceci est inhérent à l'utilisation de ce modèle, qui intervient en deuxième analyse, alors que les autres modèles ne font qu'augmenter la taille du treillis de la première analyse.

15.7.2.2 Combinaisons de modèles morpho-syntaxiques

Les modèles. Nous rappelons que les modèles facultatifs exclusivement voués à l'analyse morpho-syntaxique sont les modèles lexicaux, le modèle de casse et le modèle syntagmatique.

A ceux-ci, nous ajoutons le modèle de recherche relâchée, que nous ne considérons pas comme un véritable modèle de correction. Ce modèle est avant tout un moyen de détecter au plus tôt un type particulier de forme lexicale, comme les composés lexicaux et les IVs, lorsque ces formes ne varient de celles de nos lexiques que par des différences mineures.

Ce sont donc 5 modèles qui sont évalués ici. La Table 15.3 en présente les principales combinaisons testées.

Analyse. D'emblée, on constate que tout ajout d'un modèle améliore le taux d'étiquetage. Comme précédemment, le modèle de casse contribue énormément à l'amélioration des résultats. On constate ainsi que sa combinaison avec les modèles lexicaux seuls (12040000) confère au système sa meilleure progression.

On remarque en outre que la combinaison du modèle de casse et de la recherche relâchée (12340000) apporte une amélioration supérieure à la simple addition de leurs résultats respectifs. Ceci peut s'analyser comme suit : le modèle de casse, seul, considère tout mot OOV comme un nom propre ou un acronyme. Or, ce modèle propose à tort ces deux étiquettes pour des IVs qui ne varient de la forme recensée que par la casse. La recherche relâchée, de son côté, évite cet écueil malheureux, ce qui influence inévitablement l'analyse des catégories environnantes. Par exemple, dans la phrase « ils portent des T-shirts", l'étiquetage du mot *T-shirts* comme un nom a une influence sur l'étiquetage du mot *des*, dont l'analyse passe à juste titre de `DETPREP` [12] à celle de `DETIND` [13].

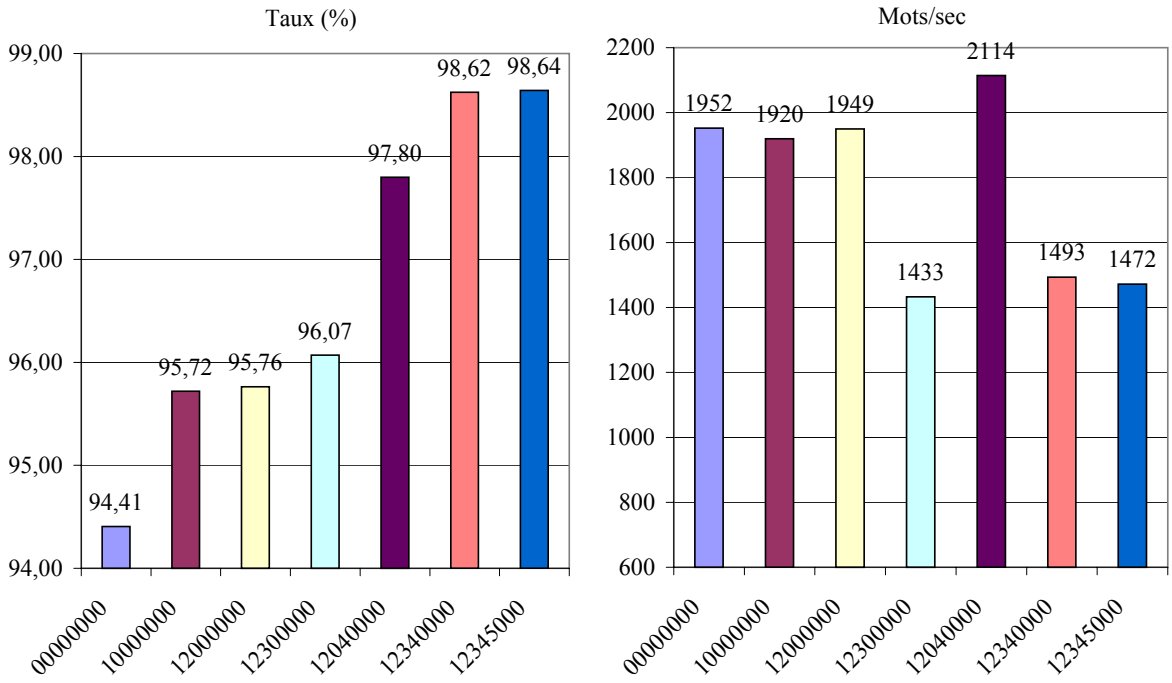
De manière générale, on peut conclure que les modèles développés dans le cadre de ce nouveau module morpho-syntaxique sont pertinents. Ensemble, ils améliorent de près de 3% les résultats que nous obtenions avec le seul modèle lexical des IVs. Le temps de traitement reste quant à lui tout à fait acceptable, puisqu'il ne descend pas en dessous des 1400 mots/sec. La combinaison de ces modèles est donc utilisable dans un système temps réel.

Le modèle de langue ainsi complété reste cependant impuissant face aux structures qui impliquent des dépendances de longue distance. Par exemple, le mot *reste* est analysé comme un nom dans « le retour vers les autres reste problématique », parce que le n-gramme *autres reste problématique* autorise l'analyse `ADJ NOUN ADJ`, fréquemment rencontrée dans le corpus, et que le verbe, *reste*, est distant de son sujet, *le retour*. Ce type de phrase nécessite incontestablement une méthode d'analyse capable de construire et d'articuler des groupes syntaxiques.

Il faut cependant constater que dans certains cas, l'analyse retenue n'est pas à proprement parler erronée. Il s'agit plutôt de cas limites. Par exemple, *tout* est analysé comme un nom, et non comme un pronom indéfini, dans la phrase « un sentiment qui ne peut empêcher le pourrissement de tout ».

Enfin, la qualité des résultats a permis de mettre en évidence des erreurs d'étiquetage du corpus. Ainsi, dans « après l'inné », *l'* était analysé comme un pronom et non comme un déterminant, tandis que *des*, dans « commencent pour les policiers des semaines et des mois » était analysé à tort `DETPREP` plutôt que `DETIND`.

[12] Contraction de l'article et de la préposition.
[13] Déterminant indéfini.



TAB. 15.3: Performances des modèles morpho-syntaxiques

15.7.2.3 Influence des modèles de correction

La Table 15.4 présente le taux d'étiquetage et la vitesse de traitement lorsque les modèles de correction sont ajoutés à l'analyse. Quelle que soit la correction intégrée, la qualité de l'étiquetage est légèrement inférieure à celle obtenue par l'analyse morpho-syntaxique seule. On constate en outre que la correction des OOVs dégrade un peu plus les résultats que la correction flexionnelle.

Quelle que soit la correction intégrée, le temps de traitement est constamment supérieur à celui de l'analyse morpho-syntaxique seule. Ceci est logique, puisque de nombreuses opérations supplémentaires sont réalisées. On constate que l'augmentation du temps de traitement est plus raisonnable en présence de la correction des OOVs qu'en présence de la correction flexionnelle. Ceci est dû au fait que la correction flexionnelle implique un *second traitement syntaxique* de la phrase.

Correction des OOVs. La dégradation de l'étiquetage est due au fait que le corpus contient environ 1 500 mots, absents de notre lexique, que le système doit malgré tout classer. De ces 1 500 mots, environ 90 sont corrigés par l'un ou l'autre des modèles de correction, tandis que les autres soit sont considérés comme des noms propres ou des acronymes, soit sont simplement trop distants des mots du lexique pour permettre aux modèles de correction de proposer des candidats. Dans l'ensemble, ce résultat met en évidence le caractère incomplet

de notre lexique, et semble indiquer que l'approche globale est saine, étant donné que le taux de fausses alarmes est relativement réduit. Ce résultat est cependant à confirmer dans le cadre de l'évaluation de la correction des OOVs présentée ci-après.

Parmi les corrections réalisées, 39 entraînent un changement de catégorie. C'est ce qui explique la légère chute des résultats en terme d'étiquetage. En voici quelques exemples :

1. Distance typographique :
   - *fortifs*, nom (abréviation de *fortifiants*) → *fortuits*, adjectif
     [insertion de u devant i, substitution de t pour f]
   - *migs*, nom → *mines*, verbe
     [substitution de n pour g, insertion de e devant s]
   - *honda*, nom propre -> *bonda*, verbe
     [substitution de b pour h]
2. Distance phonétique :
   - *réaffirmation*, nom → *réaffirmassions*, verbe.
   - *rock*, nom → *rauque*, adjectif.
   - *(lock-)out*, nom -> *aoûte*, verbe [14].

Ces corrections sont regrettables, mais on constate que le comportement des modèles développés est fidèle à ce que nous avons voulu mettre en place : une forme présentant plusieurs erreurs peut être corrigée, et les substitutions ne sont acceptées que lorsque les lettres sont contiguës sur le clavier.

Correction flexionnelle. La dégradation de l'étiquetage est provoquée par les règles autorisant les substitutions entre infinitif (*manger*), participe passé (*mangé*) et indicatif présent 2^e personne du pluriel (*mangez*), lorsque la forme concernée suit un verbe conjugué. Le système se trompe, par exemple, en insérant l'infinitif dans :

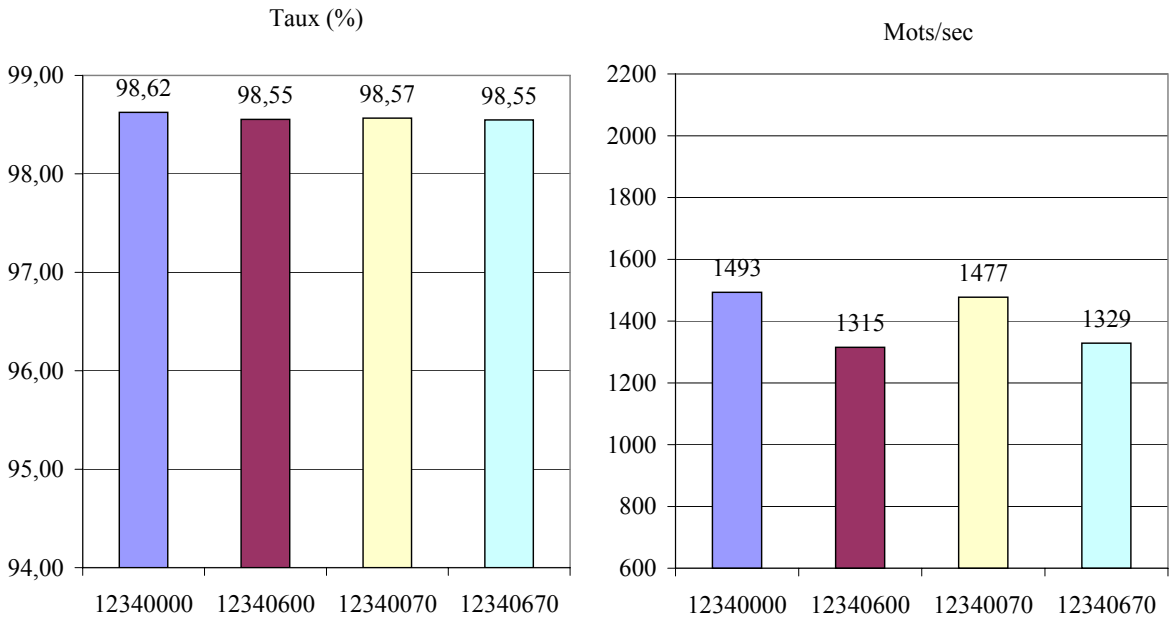
« ... ont une fois de plus prouvé » → *prouver*
« ... demeurent réservés » → *réserver*.

L'infinitif est par contre à juste titre inséré dans :

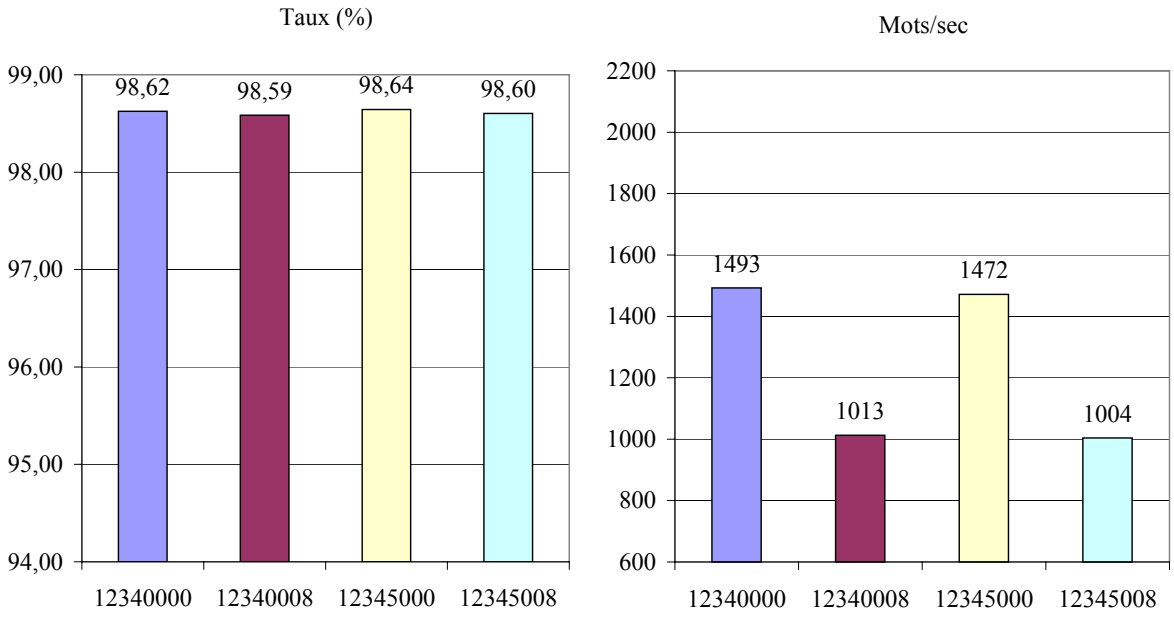
« ... ne veut plus achetez » → *acheter*.

A priori, et bien que le corpus ne soit pas exempt d'erreurs, les règles de gestion des infinitifs devraient être améliorées. Une analyse plus fine de ce résultat est proposée dans le cadre de l'évaluation de la correction flexionnelle, présentée ci-après.

[14] Du verbe *aoûter*, « Parvenir à maturation, atteindre son complet développement ».



Correction des OOVs



Correction flexionnelle

TAB. 15.4: Influence des modèles de correction

15.7.2.4 Synthèse

L'évaluation de la nouvelle analyse morpho-syntaxique montre la pertinence des nouveaux modèles qui y sont dédiés. La recherche relâchée, mais surtout le modèle de casse, améliorent considérablement l'étiquetage proposé, supérieur de 3% aux résultats de l'ancienne analyse.

Cette évaluation indique également quelques dégradations de l'étiquetage lorsque les modèles de correction sont intégrés dans le système. Bien que le corpus présente quelques coquilles, des corrections sont réalisées à tort, ce qui modifie l'étiquetage concerné. Les évaluations consacrées aux modèles de correction expliquent dans le détail les raisons de ces corrections abusives.

Le temps de traitement de l'analyse morpho-syntaxique est satisfaisant. Il souffre par contre de l'introduction des modèles de correction. Cette limite est cependant toute relative, puisque l'analyse ne descend pas en dessous des 1 000 mots/sec.

A titre de comparaison, le système de correction d'Oflazer (1996), que nous avons présenté dans l'état de l'art (cf. Section 14.4.1.4), fait état d'une vitesse de traitement, en conditions réelles, de 500 mots/sec. Il est cependant difficile de tirer une conclusion de cette information, étant donné que les temps de traitement ont été obtenus dans des conditions différentes.

Afin d'évaluer la signification du temps de traitement obtenu, notre système et le logiciel de correction Antidote, que nous avons brièvement décrit en Section 13.5.2.1, ont tous les deux été exécutés sur le même corpus et dans les mêmes conditions. Le corpus utilisé est notre corpus de test complet, comprenant plus de 66 000 mots. Notre système a été exécuté avec l'ensemble de ses modèles (configuration 12345678). Le logiciel Antidote a été exécuté en « mode accéléré » : nous l'avons limité aux corrections essentielles, en décochant toutes les options supplémentaires, comme la détection des régionalismes, ou la correction des fautes de style. Bien sûr, Antidote réalise des traitements que nous ne faisons pas. La comparaison est cependant intéressante, puisque notre système traite la totalité du corpus en 1,15 min, alors qu'Antidote le traite en 5,50 min. Notre module d'analyse morpho-syntaxique est donc relativement rapide, même lorsqu'il intègre l'ensemble de ses modèles de correction.

15.7.3 Evaluation de la correction des OOVs

Ce point commence par une présentation des trois corpus sur lesquels ont été réalisés nos tests. Sur cette base, nous décrivons les objectifs de l'évaluation, orientée *classification*. Ensuite, nous détaillons et analysons les résultats obtenus sur nos différents corpus, avant de présenter les temps de traitement, tous corpus confondus.

15.7.3.1 Corpus de test

Afin d'obtenir une évaluation significative du système de correction des OOVs, nous avons testé nos modèles sur trois corpus fort différents :

Un corpus synthétique. 229 mots corrompus ont été manuellement introduits dans 100 phrases, contenant au total 1887 mots. 12% des formes sont donc corrompues. Ces formes appartiennent toutes au lexique du système. Elles contiennent une ou plusieurs erreurs, qui se répartissent entre la casse, l'accentuation, les erreurs typographiques et les erreurs phonétiques. Les erreurs introduites sont *réalistes* dans le sens où nous les avons rencontrées dans des phrases réelles. En voici quelques exemples :

Forme correcte	**Forme erronée**
deux	*duex*
drapeau	*drapo*
gouvernement	*giuvernmènt*
guerre	*guefre*
prisonniers	*prisdonniéts*
satisfaction	*sartisgaction*
savent	*savnt*

Un corpus de presse. Ce corpus est constitué de phrases réelles, extraites d'un CD-ROM reprenant l'ensemble des articles du journal *Le Monde* de l'année 1998. Le corpus contient 725 phrases, retenues parce qu'elles présentent 1201 mots hors-lexique. Nous avons considéré comme *hors-lexique* tout mot qui ne peut être trouvé dans le lexique par simple variation de la casse. Le mot est donc soit absent du lexique, soit désaccentué et/ou corrompu.

De ces 1201 termes hors-lexique, seuls 95 sont de véritables OOVs. En voici quelques exemples :

Forme correcte	**Forme erronée**
commandantes	*comandantes*
commission	*commisson*
empuantie	*empuantée*
ému	*émû*
exécutif	*éxécutif*
flottille	*flotille*

900 des OOVs, par contre, sont des noms propres (*Anatolie*, *André*, *Dracula*, *Monet*, *Pérou*, ...), tandis que les autres sont simplement absents de notre lexique (*designers*, *mini*, *pensive*, *retirement*, *zonards*, ...).

Un corpus de mails. Nous avons également constitué un corpus de phrases extraites de mails que nous avons reçus au cours des trois dernières années. Ce corpus a été construit selon le même mode opératoire que le corpus de presse. Il regroupe 466 phrases rédigées par plus de 50 personnes différentes. Ces phrases contiennent 1069 mots hors-lexique, dont 287 véritables OOVs. Parmi ceux-ci, on trouve :

Forme correcte	**Forme erronée**
acheter	*ahceter*
avait	*avaiy*
cafétéria	*cafetaria*
constituent	*constitutent*
élève	*élèbve*
existentielle	*existencielle*
future	*futute*
quelque	*qqe*
orthographique	*orttograffic*
type	*typ*

Parmi les mots hors-lexique que nous considérons comme corrects, 700 sont des noms propres (*Artois*, *Berbère*, *Celtes*, *Ibère*, *Ligure*, ...) et quelques uns sont absents de notre lexique (*dll*, *latinisation*, *montois*, *nota bene*, *rentable*, *toner*, ...).

Les autres formes, par contre, n'appartiennent pas à la norme : ce sont des formes tronquées qui miment l'oralité (*blème*, *démo*, *dispo labo*, *probas*, ...). Nous avons cependant posé en début de chapitre (cf. 15.2.1) que le système de synthèse se doit de respecter le style et le registre choisis par l'auteur.

15.7.3.2 Objectifs

L'évaluation classiquement proposée dans le cadre de la correction des OOVs se limite généralement à estimer, sur un corpus de mots OOVs hors-contexte, le nombre de cas où le système est capable de proposer la bonne correction dans les n meilleures solutions retournées par le système. Ce mode d'évaluation, mis en œuvre par les systèmes de l'état de l'art, produit indiscutablement des scores fort élevés : les meilleurs systèmes montent à plus de 98% de correction, en considérant les 3 premières solutions proposées. Or, comme le constate Kukich (1992b), ces *solutions* ne peuvent devenir des *corrections* que si une décision est prise par le système. La prise de décision implique une seule solution, et n'assure pas que la solution retenue figure parmi les 3 premières proposées. De tels scores ne sont donc pas fort représentatifs des véritables performances du système.

Notre objectif est tout autre, parce que nous tenons compte des éléments suivants :

1. Le système de correction proposé se situe dans le cadre d'un système de synthèse de la parole. Le système de correction ne peut donc interagir avec l'utilisateur : la seule interaction du système dans son ensemble est l'émission du flux de parole correspondant au texte analysé. Le système de correction doit donc prendre des décisions de correction. Nous sommes dans le cadre d'un système déterministe.
2. La description des corpus de test a mis en évidence le fait que de nombreuses formes hors-lexique ne sont pas à corriger : il s'agit de noms propres, d'acronymes ou de formes lexicales absentes du lexique utilisé. Le système de correction des OOVs ne

peut donc corriger toutes les formes absentes du lexique. Il doit se limiter aux formes effectivement corrompues.

De ce fait, nous désirons évaluer les capacités du système à identifier les formes qu'il peut corriger, et à choisir *la bonne solution dans le contexte de la phrase*, en intégrant toutes les informations du système d'analyse. Ce processus s'apparente donc à un problème de classification : la forme est-elle à classer parmi les formes à corriger ? Dans l'affirmative, quelle correction faut-il choisir ?

Pour évaluer notre système de correction, nous nous sommes de ce fait fortement inspiré des méthodes utilisées en classification. Pour illustrer les concepts concernés, nous décrivons leur utilisation en recherche d'information.

Classification et recherche d'information. La recherche d'information est le domaine des sciences de l'information qui tâche de répondre à une requête donnée en choisissant, dans une base de données, les documents susceptibles de contenir l'information demandée. La conférence TREC est consacrée à ce vaste domaine (Voorhees & Buckland 2006).

Pour une requête donnée, la base de données contient donc deux types de documents : les documents pertinents (*Positive* ou P) et les documents non pertinents (*Negative* ou N). En fonction du résultat de la requête, cependant, les documents sont classés dans quatre catégories, comme l'illustre la Figure 15.33 :

1. Les vrais pertinents (*True Positive* ou TP), qui font partie des documents pertinents et ont été considérés comme tels par le système.
2. Les vrais non-pertinents (*True Negative* ou TN), qui font partie des documents non pertinents et ont été considérés comme tels par le système.
3. Les faux pertinents (*False Positive* ou FP), qui font partie des documents non pertinents, mais ont été considérés à tort comme pertinents par le système.
4. Les faux non-pertinents (*False Negative* ou FN), qui font partie des documents pertinents, mais ont été considérés à tort comme non pertinents par le système.

L'objectif d'un système de recherche d'information est bien sûr de maximiser le nombre des vrais pertinents, tout en minimisant le nombre des faux pertinents. Pour évaluer un tel système, de nombreuses métriques ont été proposées, dont celles-ci :

1. L'exactitude (*accuracy*), calculée comme le nombre de vrais pertinents et de vrais non-pertinents sur le nombre de documents de la base :

$$Acc = \frac{TP + TN}{P + N} \tag{15.7.3.1}$$

2. La précision, qui correspond au nombre de vrais pertinents sur le nombre de documents retournés par la requête :

$$Prec = \frac{TP}{TP + FP} \tag{15.7.3.2}$$

3. Le rappel (*recall*) ou *TPR* (*True Positive Rate*), calculé comme le nombre de vrais pertinents sur l'ensemble des documents pertinents de la base :

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{15.7.3.3}$$

4. Les fausses alarmes ou *FPR* (*False Positive Rate*), qui correspondent au nombre de faux pertinents sur le nombre de documents non pertinents de la base :

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} \tag{15.7.3.4}$$

Ces différentes métriques apportent chacune une information sur la qualité du classificateur utilisé dans le système de recherche.

Le rapport du TPR et du FPR peut en outre être projeté au sein de ce qu'on appelle l'*espace ROC* (*Receiver Operating Characteristic*) (Fawcett 2003). Dans cet espace, illustré par la Figure 15.34, la position optimale se situe au point $(0, 1)$ et représente un système capable de distinguer sans erreurs les pertinents des non-pertinents. Le point opposé, (1,0), correspond au système qui inverse systématiquement la classification. Classiquement, l'espace ROC permet de déterminer le meilleur seuil de décision pour un classificateur statistique, comme un réseau de neurones : l'espace permet de représenter l'évolution de la courbe représentant le rapport entre le TPR et le FPR, en fonction de l'évolution du seuil de décision du classificateur. Cette courbe tend généralement vers le point $(0, 1)$ avant de s'en éloigner. Le point de la courbe le plus proche de (0,1) est considéré comme le compromis idéal, et permet de retenir le meilleur seuil pour le classificateur concerné.

P	N
↓	↓
TP	FP
FN	TN

Fig. 15.33: Classification des documents d'une base de données pour une requête donnée. Différence entre la véritable pertinence des documents pour la requête, et la pertinence attribuée aux documents par le système de recherche

Application à la correction des OOVs. Notre évaluation se base sur les métriques présentées ci-dessus. Notre objectif est d'évaluer, en termes d'exactitude, de précision, de TPR et de FPR, la pertinence des modèles de correction proposés et la pertinence de différentes combinaisons de ces modèles.

Nous proposons en outre une projection du rapport entre TPR et FPR dans l'espace ROC. Cependant, pour une combinaison de modèles donnée, la projection ne consiste pas en une courbe, mais simplement en un point : nous n'avons pas fait varier le seuil de correction,

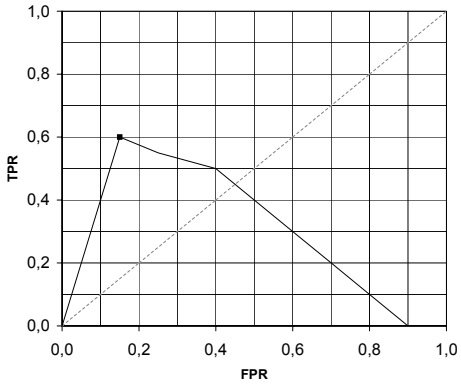


FIG. 15.34: Espace ROC

nous introduisons simplement des possibilités de correction dont la meilleure est retenue lors du calcul du meilleur chemin. La projection dans l'espace ROC a donc simplement pour objectif de déterminer la meilleure combinaison de modèles de correction.

Quel que soit le corpus, les scores présentés ne concernent que les mots *hors-lexique*. Les mots qui appartiennent à notre lexique et ne sont pas corrompus n'interviennent pas dans le calcul, de manière à ne pas artificiellement gonfler les résultats. Dans le détail :

1. *Sur le corpus synthétique*. Nous désirons estimer les performances maximales du système lorsqu'il se trouve dans les conditions idéales. C'est la raison pour laquelle aucun mot du corpus, corrompu ou non, n'est absent du lexique utilisé. Les scores sont donc calculés sur les 229 mots corrompus, qui sont tous à corriger. De ce fait, pour ce corpus

   - $P = 229$
   - $N = 0$

   Du fait des valeurs de P et N, l'exactitude et le TPR ont les mêmes valeurs, ce qui correspond bien à ce que nous voulons estimer sur ce corpus.

2. *Sur les deux corpus réels*. L'objectif de l'évaluation est principalement d'observer le comportement du système *en conditions réelles*, afin de déterminer la véritable pertinence de l'approche. En ce qui concerne le corpus de presse, les scores sont calculés sur les 1201 mots inconnus. Parmi ces mots, il y a 95 mots à corriger. Donc pour ce corpus,

   - $P = 95$
   - $N = 1106$

   Concernant le corpus de mail, les scores sont calculés sur les 1069 mots inconnus. Parmi ces mots, il y a 287 mots à corriger. Donc pour ce corpus,

   - $P = 287$

- $N = 782$

Sur ces corpus, nous nous intéressons tout particulièrement au rapport entre le TPR et le FPR dans l'espace ROC.

15.7.3.3 Résultats et analyse

La Table 15.5 présente les résultats obtenus sur les trois corpus en termes d'exactitude et de précision. La Figure 15.35 présente la projection dans l'espace ROC du rapport entre TPR et FPR.

Note 15.7.2. Dans cette section, nous donnons de nombreux exemples de correction. Les exemples prennent l'une des trois formes suivantes :

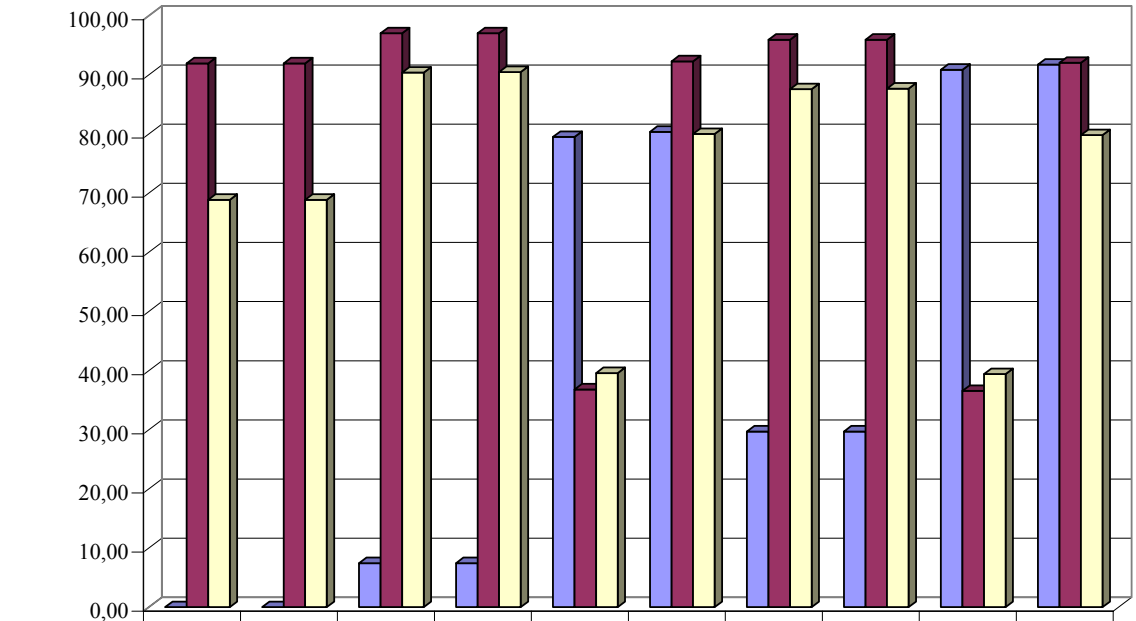
1. *forme du texte* → *forme corrigée*. Le texte qui accompagne l'exemple distingue sans ambiguïté la forme correcte.
2. *forme du texte* (*forme explicative*) → *forme corrigée*. La forme explicative ne correspond ni à la forme du texte, ni à la forme corrigée. Elle permet soit d'indiquer la forme correcte attendue, soit d'expliciter une abréviation ou une forme tronquée.
3. *forme du texte* (*forme correcte*). Dans ce cas-ci, le système de correction n'a proposé aucune correction.

Tendances générales. Le modèle de base (00000000) et les modèles lexicaux (12000000) ne sont proposés qu'à titre indicatif. Il faut cependant expliquer comment ces modèles, qui n'incluent aucun mécanisme de correction, présentent un FPR positif sur le corpus de mail. Ce FPR positif est dû au fait que ces modèles n'ont aucun moyen d'identifier les termes du lexique qui présentent une différence de casse. Parmi ces termes figure l'article élidé (*L'*), qui est considéré comme un OOV. Ceci entraîne la suppression de l'apostrophe, ce que le système d'évaluation considère comme une correction erronée.

L'introduction de la recherche relâchée entraîne un apport moins important sur le corpus synthétique que sur les corpus réels. Sur le corpus synthétique, les erreurs se limitent en fait rarement à une simple différence de casse ou d'accentuation, parce que nous avons voulu y évaluer le système sur des cas difficiles, impliquant plusieurs erreurs simultanées. Par contre, les résultats sur les corpus réels montrent que les erreurs de casse et d'accentuation sont les erreurs les plus fréquentes. La distinction entre recherche relâchée et distance typographique permet donc de gérer de nombreuses erreurs à moindre coût.

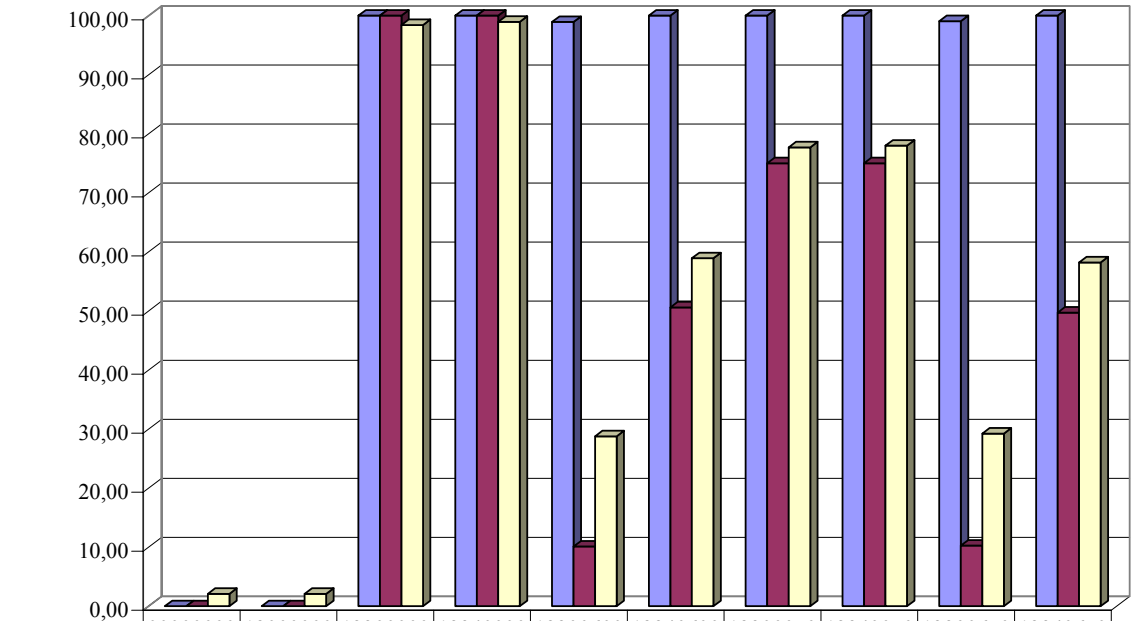
En ce qui concerne les modèles de correction :

1. En terme d'exactitude et de précision, le modèle typographique semble meilleur sur le corpus synthétique, tandis que le modèle phonétique semble plus performant sur les corpus réels.
2. En terme de TPR, le modèle typographique est systématiquement plus performant : ce modèle corrige plus. Le FPR indique par contre que ce modèle corrompt plus également.



	00000000	12000000	12300000	12340000	12300600	12340600	12300070	12340070	12300670	12340670
Synthétique	0,00	0,00	7,42	7,42	79,48	80,35	29,69	29,69	90,83	91,70
Presse	91,9238	91,9238	97,0029	97,0029	36,8031	92,2569	95,9205	95,9205	36,5533	92,0071
Mail	68,8498	68,8498	90,3653	90,4588	39,5701	79,9817	87,5589	87,6525	39,3831	79,7947

Exactitude (%)



	00000000	12000000	12300000	12340000	12300600	12340600	12300070	12340070	12300670	12340670
Synthétique	0,00	0,00	100,00	100,00	98,93	100,00	100,00	100,00	99,06	100,00
Presse	0	0	100	100	10,0966	50,6029	75,0005	75,0005	10,2507	49,7114
Mail	2,0838	2,0838	98,4215	98,9422	28,7578	58,9247	77,6983	77,9788	29,2294	58,1991

Précision (%)

TAB. 15.5: Correction des OOVs : taux et précision

3. Dans l'espace ROC, on constate que les élans du modèle typographique sont systématiquement maîtrisés lorsque ce modèle est combiné au modèle de casse. Cette combinaison est tout à fait pertinente, et propose un meilleur compromis que le modèle phonétique seul. C'est donc en combinaison avec le modèle typographique que le modèle de casse montre sa véritable utilité : en permettant un étiquetage correct des noms propres et des acronymes, il freine les excès de correction du modèle typographique.
4. L'espace ROC met également en évidence le fait que les deux modèles de correction, employés ensemble, ne sont pas totalement redondants. Les deux modèles semblent gérer des cas différents et se compléter utilement.

Cette analyse globale laisse supposer que les deux modèles de correction se complètent, mais que le modèle typographique n'est utilisable qu'en combinaison avec le modèle de casse, qui en limite les excès. La recherche relâchée, quant à elle, gère de nombreux cas qui ne nécessitent pas le déclenchement d'un processus complexe. Le meilleur compromis est donc la combinaison de l'ensemble de ces modèles (12340670), comme le montre l'espace ROC.

L'espace ROC montre en outre que les fausses alarmes sont absentes sur le corpus synthétique. Elles apparaissent par contre sur les corpus réels, le corpus de mail étant le plus touché.

Ces constats statistiques demandent confirmation au travers d'une analyse plus fine, au niveau des données de l'évaluation.

Corpus synthétique et état de l'art. De toutes les évaluations réalisées, la seule qui soit comparable aux évaluations de l'état de l'art est le taux d'exactitude obtenu sur le corpus synthétique. En effet, l'exactitude correspond aux mesures réalisées dans l'état de l'art, et les OOVs de notre corpus synthétique sont tous à corriger.

Si l'on tient compte des trois meilleures solutions proposées par le système, notre modèle corrige 98,6% des OOVs. Nous nous situons donc au niveau de l'état de l'art. De notre corpus synthétique, seuls trois cas ne sont pas gérés par le système : ils ne sont pas acceptés par le filtre restrictif, qui limite les possibilités d'édition typographique :

1. *oprelile* (*oreille*) : suppression de *p* et transposition de *il*. Ces deux opérations ne sont pas acceptées dans le même mot.
2. *inidvidujelles* (*individuelles*) : transposition de *di* et suppression de *j*. Ce cas est identique au précédent.
3. *antiracizm* (*antiracisme*) : substitution de *z* en *s* et insertion de *e*. Ces opérations sont acceptées dans le même mot, mais pas à moins de 2 caractères d'intervalle. Nous pensions que la forme serait corrigée par le modèle phonétique. Cependant, l'analyse du fichier de règles phonétiques montre que la règle concernée n'a pas été retenue, parce qu'elle compte moins de 30 occurrences dans le corpus.

Ces résultats sont appréciables, mais l'évaluation qui nous intéresse vraiment est celle réalisée en contexte. On constate dans ce cas que le système corrige 91,7% des OOVs.

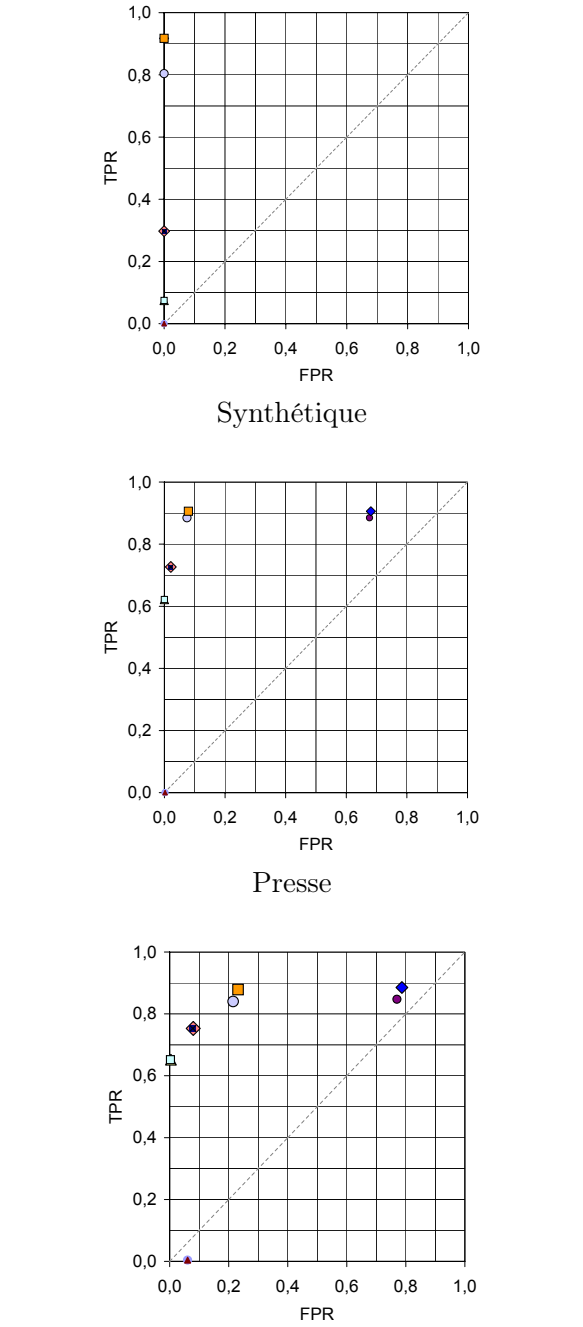


FIG. 15.35: Correction des OOVs : espace ROC

Dans des conditions idéales, c'est-à-dire en présence de formes réellement corrompues, l'approche globale propose donc une correction efficace. Les erreurs de correction sont causées par deux facteurs :

1. La forme choisie est *a priori* mieux classée que la forme correcte : *révelle* (*révèle*) → *réveille*, *virgile* (*virgule*) → *vigile*.
2. L'analyse de la forme choisie s'intègre mieux dans l'analyse de la phrase : *otres* ({*autres*, pronom}) → {*ogres*, nom}, *piries* ({*pires*, adjectif}) → {*pairies*, nom}.

Corpus de presse. Nous l'avons signalé, la majorité des erreurs de ce corpus sont des formes mal accentuées, que la recherche relâchée gère très bien : *archaiques* → *archaïques*, *fonçent* → *foncent*, *émû* → *ému*, *diner* → *dîner*. Le corpus présente cependant quelques coquilles, pour lesquelles le système propose souvent la bonne correction : *flotille* → *flottille*, *empuantée* → *empuantie* [15], *logment* → *logement*.

Dans de rares cas, la correction est erronée. Soit la forme correcte est inconnue du lexique : *epinal* (*Epinal*) → *spinal*, *modologues* (*podologues*) → *podologies*. Soit la forme choisie est plus probable ou équiprobable, mais favorisée par l'ordre alphabétique : *conjuge* (*conjugue*) → *conjuguai*.

Certaines formes ne sont pas corrigées, parce qu'elles dépassent le nombre d'erreurs permises par le filtre restrictif. C'est le cas de *prestiditateur* (*prestidigitateur*), qui présente deux suppressions côte à côte alors que le filtre, qui autorise plusieurs opérations d'édition dans un mot de cette longueur, n'accepte pas deux suppressions côte à côte. C'est aussi le cas des abréviations, comme *St* (*Saint*).

Les termes qui sont corrigés à tort sont systématiquement absents de notre lexique. Nous y distinguons :

1. Les mots qui devraient appartenir au lexique. Modifications typographiques : *retirement* → *étirement*, *pensive* → *endive*, *cannois* → *danois*, *générique* → *génétique*. Modification phonétique : *marginalisation* → *marginalisassions*.
2. Les formes tronquées : *socialo* (*socialiste*) → *social*, *restos* (*restaurants*) → *tests*.
3. Les noms propres en minuscule : *peres* ([*Shimon*] *Peres*) → *pères*.

Corpus de mails. Contrairement au corpus précédent, celui-ci contient beaucoup plus d'erreurs typographiques, très bien gérées par le modèle, qui corrige les nombreuses transpositions (*ahceter* → *acheter*, *coprus* → *corpus*, *drenière* → *dernière*, *devriat* → *devrait*, *qeu* → *que*) et insertions (*constitutent* → *constituent*, *annaotations* → *annotations*, *linguisitique* → *linguistique*, *papieur* → *papier*), ainsi que les quelques substitutions (*futute* → *future*, *avaiy* → *avait*) et suppressions (*typ* → *type*, *mignonettes* → *mignonnettes*, *ls* → *ils*).

[15] La correction peut ici surprendre, étant donné que *é* et *i* ne sont pas voisins sur le clavier. La correction est permise par la recherche relâchée, qui propose la conversion *é* → *è*. Or, le *è* est voisin du *i*. Il s'agit donc d'une séquence de deux substitutions, autorisée par l'application successive de la recherche relâchée et du modèle typographique.

Le corpus contient une seule véritable erreur phonétique, bien corrigée : *existencielle* → *existentielle*. Il comporte également quelques écritures phonétiques qui s'apparentent à de la cacographie. Le système les gère correctement : *zéducatifs* est corrigé en *éducatifs* par le modèle typographique, et *orttograffic* est corrigé en *orthographique* par le modèle phonétique.

Note 15.7.3 (Cacographie)**.** La cacographie est une orthographe fautive. Le terme, d'abord employé en pédagogie pour désigner le procédé qui consiste à enseigner l'orthographe sur la base de textes corrompus que l'élève doit corriger, fait maintenant référence au jeu de mots qui consiste à corrompre l'orthographe de chaque mot d'une phrase de la manière la plus amusante possible.

Lorsque la bonne correction appartient au treillis de solutions, il arrive que le système se trompe, mais les cas sont rares : *cherger* (*charger*) → *cherrer* [16], *pouuvoir* (*pouvoir*) → *mouvoir* [17].

Par contre, plusieurs erreurs ne sont pas ou sont mal corrigées, parce que le système *ne connaît pas* la solution. Nous y discernons 4 types différents :

1. Les mots considérés comme des noms propres du fait de la majuscule en début de mot : *Poiur* (*pour*), *Recomandation* (*recommandation*).
2. Les abréviations. Certaines ne sont pas corrigées : *bcp* (*beaucoup*), *qqe* (*quelque*). D'autres sont mal corrigées : *tjs* (*toujours*) → *tus*, *bb* (*bébé*) → *bob*, *stp* (*s'il te plaît*) → *stop*.
3. La répétition de lettres à usage expressif : *grrrrrrrrroooos* (*gros*).
4. Les mots inconnus du lexique : *atonium* (*Atomium*) → *actinium*, *chartente* (*Charente*) → *chargent*.

Enfin, de nombreuses *formes correctes* sont modifiées à tort, parce qu'elles n'appartiennent pas au lexique. 5 classes peuvent y être distinguées :

1. Les mots qui devraient appartenir au lexique. Modifications typographiques : *doctorants* → *doctorats*, *rentable* → *retable*, *portable* → *potable*, *auditée* → *usitée*, *détournement* → *défournement*, *bénéfiques* → *génétiques*. Modifications phonétiques : *latinisation* → *latinisassions*, *municipalisation* → *municipalisassions*, *romanisation* → *romanisassions*, *gym* → *geins*.
2. Les mots étrangers. Modifications typographiques : *(nota) bene* → *benz*. Modifications phonétiques : *review* → *revues*, *toner* → *tonnerre*, *bye* → *baille*.
3. Les noms propres en minuscule. Modifications typographiques : *ibère* → *bière*, *berbère* → *bergère*, *marcourt* → *parcourt*, *montois* → *mongols*. Modifications phonétiques : *alaman* → *allemands*, celtes → cultes.

[16] *Cherrer* : « Dépasser un prix, vendre cher ».

[17] Le lexique contient bien *pouvoir* en tant que nom ou infinitif. Cependant, le poids attribué à *pouvoir* en tant qu'infinitif désavantage cette forme par rapport à *mouvoir*, qui ne peut être qu'infinitif et a de ce fait un poids moins élevé, malgré la distance typographique supérieure.

4. Les abréviations et mots tronqués. Modifications typographiques : *démo* (*démonstration*) → *démon*, *probas* (*probabilités*) → *robés*, *fac* (*faculté*) → *face*, *max* (*maximum*) → *mas*. Modification phonétique : *resto* (*restaurant*) → *rustaud*.
5. Les termes spécialisés. Nous n'avons rencontré que des modifications typographiques : *dll* → *dol*, *ffr* → *fer*, *softs* → *sorts*.

15.7.3.4 Temps de traitement

Etat de l'art. L'un des objectifs de l'évaluation est de comparer nos temps de traitement à ceux des systèmes de l'état de l'art qui fournissent cette information : l'automate tolérant aux erreurs d'Oflazer (1996), et l'automate de Levenshtein de Schulz & Mihov (2002) (cf. Section 14.4.1.4). Les résultats de ces deux systèmes sont rappelés en Table 15.6.

Méthodologie. Cette évaluation a été réalisée sur l'ensemble des formes, extraites de nos 3 corpus de test, qui contiennent effectivement une ou plusieurs erreurs : 611 formes au total $(229 + 95 + 287)$.

Pour un OOV donné, le temps de traitement évalué correspond à la composition filtrée entre l'OOV et le dictionnaire, en ce compris la projection de la sortie du résultat. Les filtres évalués sont :

1. La distance typographique :
   $OOV \circ (Long \circ Edit) \circ Lexicon$
2. La distance typographique combinée à la recherche relâchée :
   $OOV \circ (Match \circ Long \circ Edit) \circ Lexicon$
3. La distance phonétique :
   $OOV \circ Phonet \circ Lexicon$

Nous rappelons que nos modèles ne travaillent pas en fonction d'un nombre fixe d'erreurs : le modèle typographique fait varier le nombre d'erreurs en fonction d'intervalles de longueur, tandis que le modèle phonétique tient compte du contexte graphémique pour convertir un graphème en un autre, phonétiquement équivalent. La recherche relâchée ajoute en outre au modèle typographique toutes les variations d'accent et de casse possibles.

Afin de malgré tout favoriser la comparaison de notre système à ceux de l'état de l'art, nous avons déterminé les temps de traitement en fonction des intervalles de longueur du modèle typographique :

1. De 1 à 5 caractères : 1 erreur.
2. De 6 à 10 caractères : 2 erreurs.
3. De 11 à 15 caractères : 3 erreurs.
4. Plus de 15 caractères : 4 erreurs.

Système	1 erreur	2 erreurs	3 erreurs
Oflazer (1996)	15	95	349
Schulz & Mihov (2002)	2,2	30	160

TAB. 15.6: Temps de traitement (en ms) des systèmes de l'état de l'art

Modèle	1 à 5 car.	5 à 10 car.	11 à 15 car.	+ de 15 car.	Moyenne
Distance typographique	3,33	10,02	23,23	62,50	12,22
Distance typographique + Recherche relâchée	6,11	17,43	39,59	92,50	20,64
Distance phonétique	0,45	0,17	0,45	2,50	0,20

TAB. 15.7: Temps de traitement (en ms) des différents modèles de correction

La comparaison est cependant limitée parce que :

1. La distance phonétique n'est pas limitée par un nombre d'erreurs, mais par le contexte graphémique dans lequel les règles peuvent être appliquées. La répartition des résultats de la distance phonétique en fonction de la longueur des mots n'est donc pas significative.
2. Les systèmes de l'état de l'art acceptent le même nombre d'erreurs quelle que soit la longueur du mot. Les résultats qu'ils obtiennent concernent donc l'ensemble des formes de leur test, alors que nos résultats sont obtenus sur des sous-ensembles de notre corpus de test.
3. Les tests ont été réalisés sur du matériel différent.
4. Les langues des différentes évaluations varient en fonction des systèmes. Oflazer (1996) a évalué son approche en français, tandis que Schulz & Mihov (2002) ont évalué la leur en allemand.

Résultats et analyse. La Table 15.7 présente les résultats obtenus. Dans l'ensemble, on constate que les temps de traitement sont tout à fait raisonnables. La distance phonétique est de loin le modèle le moins coûteux.

La recherche relâchée, de son côté, augmente considérablement le temps de traitement nécessaire. Cependant, les résultats de l'évaluation qualitative ont montré combien ce modèle est pertinent. Le temps de traitement somme toute réduit démontre que ce modèle est également utilisable.

La comparaison aux systèmes de l'état de l'art semble favoriser notre approche. Il est en tout cas incontestable que le fait de faire varier le nombre d'erreurs en fonction de la

longueur du mot permet de considérablement limiter le temps de traitement. L'évaluation qualitative a en outre mis en évidence le fait que cette adaptation à la longueur produit un taux de correction très élevé. L'adaptation automatique à la longueur favorise donc le temps de traitement, sans nuire à la qualité des résultats.

On constate que lorsque le système est limité à une seule erreur, notre modèle typographique semble légèrement plus lent que celui de Schulz & Mihov (2002). Ceci demande une explication. Le nombre de candidats retournés pour un OOV donné est inversement proportionnel à sa longueur : pour un même nombre d'erreurs, un mot court génère beaucoup plus de candidats qu'un mot long. Par exemple, le mot *anticonstitutionnellement* ne produit qu'un seul candidat, lui-même, alors que le mot *mille* génère 16 candidats. Ceci est probablement dû au fait que le lexique contient bien plus de mots courts que de mots longs. Notre évaluation limitée à 1 erreur ne tient compte que des mots de maximum 5 caractères, alors que celle de Schulz & Mihov (2002) tient compte de l'ensemble du corpus utilisé. On peut à juste titre supposer que les mots moyens et longs favorisent le temps de traitement global de ce système lorsqu'une seule erreur est acceptée.

15.7.3.5 Synthèse

L'analyse des données confirme l'interprétation des graphes statistiques : tous les modèles contribuent à un système équilibré, où les modèles de correction ne sont autorisés à intervenir que lorsque la recherche relâchée et le modèle de casse ne peuvent proposer de solution. La séparation de la recherche relâchée et de la distance typographique permet en outre de corriger de nombreuses formes sans mettre en œuvre un processus lourd qui pénaliserait inutilement le temps de traitement.

Dans l'ensemble, le modèle typographique est plus efficace que le modèle phonétique : il corrige à lui seul de nombreuses erreurs. Le modèle phonétique contribue cependant au système, en résolvant des cas que le modèle typographique ne gère pas.

Le modèle de correction typographique semble *suffisant*, étant donné le peu de cas qui présentent des erreurs typographiques qu'il ne peut gérer : ce modèle gère un nombre d'erreurs variable en fonction du type de l'opération et de la longueur du mot. En outre, la combinaison avec la recherche relâchée accroît encore les possibilités du modèle. En nombre de corrections proposées, le modèle typographique dépasse donc indiscutablement l'état de l'art, où la distance d'édition ne permet qu'une seule opération par mot. Le principe du filtre de composition est donc pertinent.

On peut à juste titre apprécier l'excellent comportement du modèle phonétique, qui propose des homophones parfaits (*existencielle*/*existentielle*, *municipalisation*/*municipalisassions*), et parvient à rapprocher des formes étrangères sémantiquement proches (*review*/*revues*), ainsi que des formes étymologiquement liées (*alamans*/*allemands*). Il est simplement malheureux que la plupart des corrections proposées soient à rejeter, parce que la forme du texte est correcte, mais n'appartient pas à notre lexique...

Deux types de cas ne sont pas gérés par notre système :

1. Les abréviations (*bcp*, *qqe*). Ces formes ne peuvent rester en l'état, parce qu'elles doivent être prononcées par le système de synthèse, et doivent être comprises par les auditeurs du système. Il faut donc un mécanisme d'expansion des abréviations.
2. Les formes tronquées (*fac*, *fortifs*, *resto*). Ces formes doivent rester telles quelles, parce qu'elles font partie du style de l'auteur. Le système doit pouvoir les analyser et les phonétiser.

Nous terminons enfin par un constat de première importance : un système qui se fonde sur un lexique, pour orienter le traitement, *doit* posséder un lexique de qualité. Ce n'est certainement pas le cas du lexique que nous employons, au vu du nombre de termes qui en sont absents...

15.7.4 Evaluation de la correction flexionnelle

15.7.4.1 Objectifs et corpus de test

Comme dans le cas de la correction des OOVs, nous désirons évaluer les capacités du système à identifier les formes qu'il peut corriger, et à choisir *la bonne solution dans le contexte de la phrase*, en intégrant toutes les informations du système d'analyse. Notre objectif est donc d'évaluer, en termes d'exactitude, de précision, de TPR et de FPR, la pertinence des modèles de correction proposés et la pertinence de différentes combinaisons de ces modèles.

La difficulté de cette évaluation a été de se procurer un corpus réel contenant un panel représentatif des erreurs flexionnelles susceptibles de se produire dans un texte. De ce fait, nous avons *construit* un corpus que nous qualifions de *semi-synthétique*. Le corpus de base est constitué de 200 phrases extraites d'articles de presse. Chaque phrase a été choisie parce qu'elle contenait au moins une erreur. Cependant, afin de tester le système en présence d'erreurs multiples, nous avons ajouté des erreurs supplémentaires. Le principe que nous avons respecté a été de n'ajouter que des erreurs que nous avions effectivement rencontrées dans des textes. Deux types d'erreurs ont été introduites :

1. Des erreurs flexionnelles, ajoutées à proximité d'une forme déjà erronée. Dans la plupart des cas, les formes erronées sont contiguës. L'objectif est de tester la capacité du système à corriger les formes erronées plutôt que les formes correctes qui les entourent.
2. Des erreurs typographiques ou phonétiques, introduites soit dans un mot présentant déjà une erreur flexionnelle, soit dans un mot contigu à un mot présentant une erreur flexionnelle. L'objectif est de vérifier que la correction flexionnelle traite indifféremment toute forme correcte, qu'elle soit issue ou non de la correction des OOVs.

Pour le système d'évaluation, une forme erronée constitue une seule erreur, même si la forme cumule différents types d'erreurs. Sur cette base, les 200 phrases du corpus, contenant 3 044 mots, présentent 344 erreurs. Ces erreurs, qui couvrent donc 11,3% du corpus, sont réparties comme suit :

1. 333 erreurs flexionnelles simples.
2. 10 formes présentant 1 erreur flexionnelle et 1 erreur d'édition.
3. 1 forme ne contenant qu'une erreur d'édition, mais située à côté d'une forme présentant une erreur flexionnelle.

La Table 15.8 propose un échantillon des erreurs flexionnelles du corpus, tandis que la Table 15.9 présente l'ensemble des formes qui cumulent erreurs flexionnelles et erreurs d'édition.

Une différence majeure distingue la correction flexionnelle de la correction des OOVs. Si la correction des OOVs ne concerne que les formes qui n'appartiennent pas au lexique du système, la correction flexionnelle concerne par contre *toutes les formes de la phrase*. En effet, toute flexion est *a priori* suspecte, et la décision de déclencher le processus de correction d'une flexion donnée dépend uniquement de son adéquation au contexte qui l'entoure. Ceci a une influence sur la *portée* de l'évaluation. En correction des OOVs, l'estimation ne tenait compte que des OOVs. En correction flexionnelle, l'estimation doit tenir compte de tous les termes de la phrase. Ceci implique que sur notre corpus de 3 044 mots contenant 344 erreurs flexionnelles,

- $P = 344$
- $N = (3\,044 - 344) = 2\,700$

15.7.4.2 Résultats et analyse

La Table 15.10 présente les taux d'exactitude et de précision. La Figure 15.36 présente le rapport du TPR et du FPR projetés dans l'espace ROC.

Tendance générale. Les combinaisons 12340000 et 12345000 ne sont présentées qu'à titre indicatif : ils n'incluent aucun mécanisme de correction flexionnelle, et servent exclusivement de point de repère. Leur exactitude correspond à l'inverse du taux de formes corrompues dans le corpus. Leur précision, quant à elle, est faussée par la portée de l'évaluation : tous les termes du corpus sont pris en compte, ce qui explique cette précision de 100%. L'espace ROC rend mieux compte de l'insuffisance de ces modèles, puisque le rapport de leur TPR et de leur FPR se situe au point $(0, 0)$.

Le modèle flexionnel semble efficace : le rappel (TPR) est de 66,32%, pour un taux de fausses alarmes (TPR) d'à peine 1,69%. La projection dans l'espace ROC montre d'ailleurs que ce modèle est performant, quelle que soit la combinaison.

Sur le corpus de test, la présence du modèle syntagmatique ne modifie que fort peu les performances des modèles de correction. Les résultats des combinaisons 12340678 et 12345678, par exemple, sont strictement identiques.

La présence des modèles de correction des OOVs, par contre, augmente de près de 4% le TPR (62,84% → 66,32%) sans considérablement modifier le FPR (1,5% → 1,69%).

Erreurs	Corrections
1 flexion	
... le salut viens ...	vient
... la réputation fragiles ...	fragile
... se faire aimé ...	aimer
... auxquels se livre tous les écoliers ...	livrent
... sans jamais quittez leur mitraillette ...	quitter
... une tel situation ...	telle
... on aimerais que ...	aimerait
... les paysans qui y travaille ...	travaillent
... il existais ...	existait
... notre après-guerre a aligner ...	aligné
2 flexions à proximité	
... les arbres doive resté ...	doivent rester
... l'ordre dans lequel ces textes se suive avait été dicter par ...	suivent ... dicté
... les professeur font chanté ...	professeurs ... chanter
... qui l'a décider à achetés ce paquet ...	décidé ... acheter

TAB. 15.8: Correction flexionnelle : erreurs flexionnelles

Erreurs	Corrections
flexion et édition typographique	
... exerce sur lui l'attriats ...	attrait
... les écoles aux mur calcfinés ...	murs calcinés
... il faut donc salué comme il conbiens ...	saluer ... convient
... j'avait inviteer ...	avais invité
... la grille qui dveais être ...	devait
... le gouvernement qui peut se flaté ...	flatter
flexion et édition phonétique	
... a une bonne idée de rassamblé ...	rassembler
... se faire respekté ...	respecter
... sur les cantaine de bases ...	centaines
... des frase courtes ...	phrases
... où l'on privilégeiais l'apprentissage ...	privilégiait

TAB. 15.9: Correction flexionnelle : erreurs flexionnelles et erreurs d'édition

Ceci indique que les erreurs flexionnelles peuvent être corrigées lorsque les termes présentent des erreurs d'édition, pour autant que la correction des OOVs fasse partie des modèles utilisés.

Ces constats statistiques, fort positifs, doivent cependant être complétés d'une analyse des données de l'évaluation.

Couverture. Les corrections réalisées par le système couvrent l'ensemble des erreurs que nous avons modélisées. Nous rappelons que les grands points de cette modélisation sont l'accord entre le déterminant, l'adjectif et le nom, l'accord entre le sujet et le verbe (que le sujet soit un pronom ou un nom), la gestion de l'infinitif après un verbe conjugué ou une préposition et la gestion du participe après un auxiliaire. A titre indicatif, toutes les erreurs de la Table 15.8 sont corrigées par le système.

Erreurs d'édition. En présence des modèles de correction des OOVs, toutes les erreurs incluant une erreur d'édition (cf. Table 15.9) ont été corrigées. Ceci indique simplement que le système traite de manière identique l'ensemble des formes correctes, qu'elles soient issues ou non de la correction des OOVs. En l'absence de ces modèles, aucune des erreurs flexionnelles liées à une erreur d'édition n'est corrigée. Ceci est cohérent avec le modèle proposé.

On constate que la correction des OOVs modifie à tort quelques formes du corpus qui n'appartiennent pas à son lexique (*communautaire* → *comminatoire*, *privatiseurs* → *privatisera*). Ces modifications sont la raison de la légère augmentation du FPR en présence des modèles de correction des OOVs. Elles montrent encore une fois la nécessité d'un lexique de qualité pour réaliser une correction efficace, et non corruptrice.

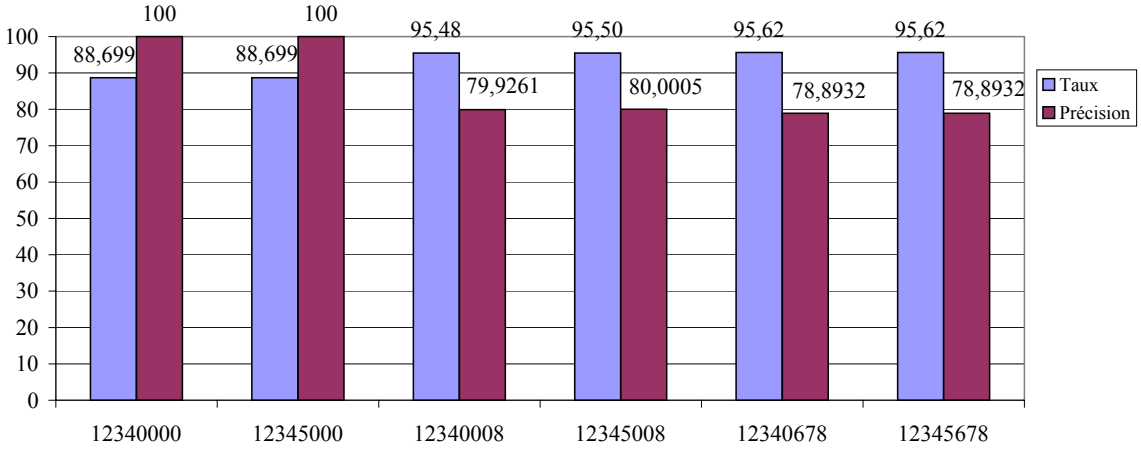
Les erreurs non corrigées. Trois causes différentes expliquent l'incapacité du système à corriger certaines erreurs du corpus :

1. *La probabilité de la correction dans le modèle de flexion.* La correction est introduite dans le treillis de solutions, mais n'est pas conservée par le modèle de flexion, qui lui préfère la forme originale :

   cet réputation fut relancé
   → *cet**te** réputation fut relancé*

   Les données utilisées pour l'entraînement sont probablement insuffisantes.

2. *Le modèle de langue.* Une correction est proposée, mais est erronée, parce que la forme à corriger a été mal analysée par le modèle de langue. La catégorie retenue empêche donc la correction désirée :



TAB. 15.10: Correction flexionnelle : taux et précision

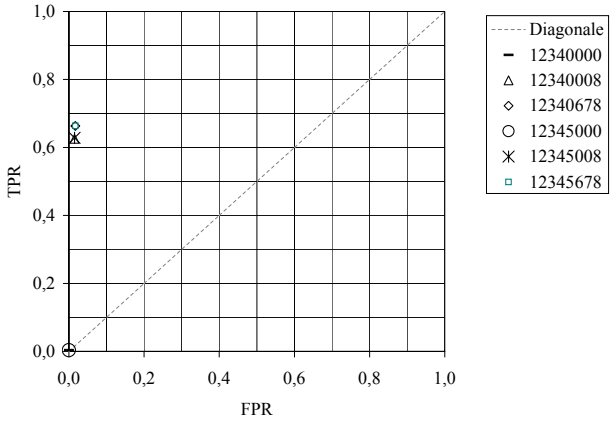


FIG. 15.36: Correction flexionnelle : espace ROC

qui le saisi<u>s</u> (verbe)
→ *qui le saisi<u>i</u>* (nom)

la guérilla antimarxiste entretien<u>s</u> (verbe)
→ *la guérilla antimarxiste entret<u>ien</u>* (nom)

3. *Les dépendances de longue distance.* Le modèle de flexion ne peut les gérer, parce qu'elles sortent de la fenêtre du n-gramme :

les hommes s'adress<u>e</u> à eux ... et tap<u>e</u>...
→ *les hommes s'adresse**nt** à eux ... et tap<u>e</u>...*

Les modifications erronées. Il arrive au modèle de modifier à tort la flexion d'une forme correcte. Les causes de ces modifications superflues sont les suivantes :

1. *Plusieurs propositions de correction.* Le système de détection propose plusieurs solutions, et le modèle flexionnel choisit la plus probable, qui n'est malheureusement pas la correction attendue :

les hau<u>t</u> quartiers
→ *l<u>e</u> hau<u>t</u> quartie<u>r</u>*

2. *Le manque d'informations morphologiques.* Les adverbes collectifs (*beaucoup de*, *plusieurs*) et les nombres ne présentent aucun trait grammatical dans notre lexique. Lorsqu'une erreur survient dans leur entourage, le système manque d'information et propose une mauvaise correction :

beaucoup de romancie<u>r</u> intelligents
→ *beaucoup de romancie<u>r</u> intelligen<u>t</u>*

plusieurs kilomètres carr<u>é</u>
→ *plusieurs kilomètr<u>e</u> carr<u>é</u>*

trois grandes cas<u>e</u>
→ *trois grand<u>e</u> cas<u>e</u>*

3. *Le modèle de langue.* Comme dans le cas des erreurs non corrigées, le modèle de langue retient la mauvaise catégorie syntaxique, ce qui induit en erreur le modèle flexionnel :

j'avait inviteer ma famille plus (adverbe) *l'ami de Yves*
→ *j'avais invité ma famille plut* (verbe) *l'ami de Yves*

4. *Les dépendances de longue distance.* Les sujets coordonnés ne peuvent être gérés par le modèle, qui accorde le verbe avec le second sujet lorsque celui-ci est au singulier :

le plaisir et le partage paraissent hors de portée
→ *le plaisir et le partage paraisse hors de portée*

15.7.4.3 Synthèse

Le modèle de correction flexionnelle que nous avons proposé se divise en trois parties distinctes : détection des erreurs flexionnelles, génération des flexions en fonction des erreurs détectées, et choix de la flexion à l'aide du modèle flexionnel. L'ensemble est cohérent et produit des résultats intéressants sur le corpus de test que nous avons constitué : le rapport entre TPR et FPR montre que le système corrige plus qu'il ne corrompt.

Certaines des erreurs du modèle peuvent être corrigées : les lexiques morphologiques peuvent être complétés, et de nouvelles règles de détection peuvent être ajoutées au modèle.

Par contre, les erreurs dues au modèle de langue ou au modèle flexionnel, de même que les erreurs liées aux dépendances de longue distance, mettent en évidence la limite de l'approche proposée : les modèles statistiques ne sont pas suffisants pour une correction flexionnelle de qualité, parce qu'ils ne détiennent qu'une vision fort partielle de la phrase. Le modèle de correction manque d'informations relatives à l'agencement de la phrase en *syntagmes*.

Le modèle, en l'état, semble donc difficilement applicable dans le cadre d'un système déterministe. Il est probablement nécessaire de reconsidérer le modèle de détection, de même que le modèle flexionnel.

Le modèle de génération flexionnelle est par contre tout à fait pertinent : les nouvelles flexions sont introduites directement dans le graphe représentant la phrase, sans devoir traiter les formes individuellement. Ceci est dû à la présence des marqueurs, qui facilitent considérablement la mise en œuvre de ce type de tâches. Le modèle proposé présente donc l'avantage certain de reposer exclusivement sur des machines à états finis. Les diverses informations relatives aux formes de la phrase peuvent ainsi être conservées dans ce format, et complétées au besoin, en attendant que la décision finale soit prise.

15.8 Avant de conclure

Le modèle de correction que nous présentons au Chapitre 16 se fonde en grande partie sur les mêmes postulats et les mêmes outils que le modèle que nous venons de présenter. Nous proposons de ce fait une conclusion commune aux deux modèles au Chapitre 17.

Chapitre 16

Correction en scènes naturelles

16.1 Introduction

La démocratisation des prix des appareils photos numériques de toutes qualités, depuis le matériel d'entrée de gamme jusqu'au matériel professionnel, a rendu la prise de photos numériques à la portée de tous. Ceci a ouvert la voie au développement de nombreuses applications visant à extraire – voire à *comprendre* – le texte présent dans ces images. Parmi ces applications, figurent en très bonne place les systèmes d'aide aux personnes visuellement handicapées, comme le classement automatique de documents ou la lecture automatique du courrier, qui reposent sur des systèmes de reconnaissance des caractères.

Cependant, de nombreux facteurs rendent souvent difficile la reconnaissance des textes présents dans ces images. L'image peut en effet présenter de nombreuses dégradations, du fait de problèmes de luminosité, d'exposition ou de focus, ou à cause de la résolution choisie ou de capteurs bruités qui, par exemple, rendent les couleurs imprécises. Le cadrage peut en outre être à l'origine de caractères coupés ou de mots tronqués. Enfin, l'orientation du texte peut être en cause, mais également la variété des polices de caractères utilisées, dont certaines présentent des effets artistiques prononcés. L'ensemble de ces caractéristiques, que l'on retrouve rarement dans les documents numérisés à l'aide de scanners à plat, classent les images prises par les appareils photos numériques dans la catégorie des *scènes naturelles*.

Définition 16.1.1 (Scène naturelle)**.** *En traitement d'images, une* scène naturelle *est une image photo ou une trame de film vidéo prise dans le monde réel, sans aucune contrainte.*

Confronté à une scène naturelle, le système de reconnaissance n'a aucune connaissance *a priori*, ni aucune information fiable concernant l'environnement, les conditions d'éclairage, ni la disposition des objets ou du texte dans l'image. Un système de reconnaissance des caractères, même lorsqu'il est dédié aux scènes naturelles, est donc plus enclin aux erreurs que les systèmes de reconnaissance utilisés dans les scanners à plat. Une étape de correction après l'étape de reconnaissance est dès lors nécessaire.

Contexte. L'application qui accueille le modèle de correction proposé est un assistant mobile de lecture pour personnes aveugles et malvoyantes (Gaudissart *et al.* 2005), développé dans le cadre d'un projet de recherche financé par le Ministère de la Région wallonne.

Cette application s'inscrit dans le cadre général du traitement des scènes naturelles : installée sur un assistant personnel (PDA [1]), elle permet à l'utilisateur de prendre des photos, dont les zones de texte sont extraites et lues automatiquement par un système de synthèse de la parole. Des exemples concrets d'utilisation de cette application sont l'identification de correspondants sur le courrier traditionnel, la lecture d'étiquettes dans les supermarchés ou la reconnaissance des valeurs sur les billets de banque. L'objectif global de l'application est donc de rendre les personnes aveugles et mal voyantes plus autonomes. Un site internet présente les principaux résultats du projet [2].

Dans le cadre général de cette application, nous avons collaboré avec une chercheuse du projet, spécialisée en reconnaissance de caractères dans les scènes naturelles (Thillou *et al.* 2005, Mancas-Thillou 2006). L'objectif de la collaboration était de rendre l'application robuste aux erreurs de reconnaissance, en intégrant un modèle de correction flexible, évolutif et portable sur plateforme embarquée.

Il est important de noter que le système de synthèse utilisé dans cette application n'est pas eLite, parce que notre synthétiseur n'est pas encore complètement portable sur plateforme embarquée. Le modèle de correction est de ce fait volontairement limité à la correction des OOVs, afin d'éviter une double analyse morpho-syntaxique sur une plateforme aux ressources limitées.

Les résultats de cette collaboration ont été présentés dans (Beaufort & Mancas-Thillou 2007) à ICDAR'07, une conférence spécialisée dans l'analyse et la reconnaissance de documents.

Ce chapitre est organisé comme suit. La Section 16.2 fait le point sur l'origine des erreurs en reconnaissance de caractères. Nous présentons ensuite, en Section 16.3, un rapide survol des quelques systèmes OCR qui intègrent de la correction. Sur cette base, la Section 16.4 décrit dans le détail le modèle de correction proposé. Nous analysons ensuite en Section 16.5 les performances du système en termes de taux de correction et de temps de traitement. La Section 16.6 referme ce chapitre, sur une synthèse de la méthode.

16.2 Erreurs et postulat

Un système de reconnaissance de caractères se divise classiquement en deux modules. Le premier est le système de segmentation, qui détecte le texte dans l'image et le segmente en caractères. Le second est un classificateur, qui attribue une étiquette à chaque caractère segmenté. Dans l'ensemble, les erreurs du système ont donc deux origines possibles :

[1] PDA : *Personal Digital Assistant.*
[2] `tcts.fpms.ac.be/projects/sypole`.

1. Le système de segmentation, qui a mal estimé la position d'une frontière entre deux caractères. Dans ces conditions, le classificateur est dans l'incapacité de proposer une classification correcte. Une erreur de segmentation aboutit par exemple à la confusion des paires $\{m, rn\}$ et $\{li, h\}$.
2. Le classificateur lui-même, qui classe difficilement certaines paires de caractères lorsque l'image est dégradée. Une erreur courante, par exemple, est la confusion de la paire $\{i,l\}$.

Le module de correction doit donc s'inscrire à la suite d'un module de reconnaissance susceptible de produire des erreurs où les classes attendues et les classes choisies par le système ne sont pas toujours alignées : nous sommes dans le cas d'erreurs $n \rightarrow m$.

Postulat 16.2.1 (Erreurs $n \rightarrow m$)**.** *Une gestion des erreurs $n \rightarrow m$, indépendante du contexte, est une nécessité en reconnaissance des caractères.*

16.3 Survol de la correction en scènes naturelles

Nous rappelons que nous avons dressé, au Chapitre 14, un état de l'art de la correction orthographique, tous domaines confondus, qui s'appuie sur de nombreuses références appartenant au domaine de la reconnaissance de caractères. Nous ne proposons de ce fait ici qu'un rapide survol des quelques méthodes dédiées à la reconnaissance de caractères et qui se distinguent de l'état de l'art précédent.

Systèmes sans correction. Dans le domaine de la reconnaissance de formes, les systèmes évitent fréquemment le recours à tout modèle de correction. L'amélioration des taux de reconnaissance est classiquement obtenue par multiplication des classificateurs : différentes sources de calcul confortent ou infirment la classification du reconnaisseur initial. C'est le cas, par exemple, de l'approche proposée par Lopresti & Zhou (1997) pour le traitement des images Web.

Les rares recours aux ressources linguistiques sont le fait de systèmes commerciaux, qui ne réalisent cependant aucune correction : un dictionnaire est consulté, mais uniquement afin de déterminer les mots hors-vocabulaire pour lesquels l'aide de l'utilisateur doit être sollicitée. C'est le cas par exemple du système de Zhang *et al.* (2002), dédié à la reconnaissance des signes chinois. Le recours à l'utilisateur est cependant parfois inapproprié et peut rendre l'application inutilisable. C'est le cas en synthèse de la parole, mais c'est également le cas si l'utilisateur est visuellement handicapé. Une correction automatique est dès lors nécessaire.

Systèmes avec correction. Dans le domaine spécifique de l'OCR, les méthodes de correction les plus légères ignorent le dictionnaire et utilisent des approches probabilistes et des modèles de langue, classiquement résolus au travers de HMMs (Borovikov *et al.* 2004) ou par

programmation dynamique (Neuhoff 1975). Sur cette base, Thillou *et al.* (2005) ont proposé d'exploiter les 3 meilleures sorties de l'OCR pour construire un tri-gramme de lettres résolu par programmation dynamique. Cependant, malgré le recours aux trois meilleures solutions de l'OCR, l'absence d'un dictionnaire est la source de deux inconvénients : premièrement, certains tri-grammes peu probables sont corrigés à tort, ce qui transforme un mot réel et correct en un OOV, et deuxièmement, les mots présentant plusieurs erreurs sont très difficiles à corriger.

Les dictionnaires ont cependant été utilisés dans certains systèmes pour la correction des OOVs en OCR. La plupart des méthodes (Bunke 1995, Jones *et al.* 1991) s'inscrivent dans la lignée de notre état de l'art général. La méthode proposée par Kolak *et al.* (2003), par contre, s'en distingue. Cette approche se base sur un entraînement au cours duquel deux modèles sont construits. L'objectif du premier modèle est de diviser chaque mot du dictionnaire en ses deux sous-séquences de caractères les plus probables : par exemple,

$$example \Rightarrow \begin{cases} ex \mid ample \\ exam \mid ple \end{cases}$$

L'objectif du second modèle est de proposer, à partir du corpus d'entraînement, une liste de séquences corrompues susceptibles de correspondre à une sous-séquence d'un mot : par exemple,

$$exam \Rightarrow exain, cxam, \text{ etc.}$$

Les deux modèles sont composés ensemble sous la forme d'un seul transducteur. Ce transducteur est ensuite inversé, de manière à accepter en entrée des mots corrompus, et à produire en sortie des mots valides. Cet unique transducteur génère en sortie un treillis de toutes les séquences de mots valides pour une séquence de caractères reçue en entrée. L'approche donne de bons résultats, mais présente deux inconvénients. Le premier est la forte dépendance du modèle de confusion au contexte : un caractère qui serait confondu dans un contexte différent de ceux observés sur le corpus ne serait pas identifié. Le second inconvénient est certainement le fait que les deux modèles sont composés en un seul transducteur : si le processus comportait une étape pour chaque modèle de l'entraînement, il serait possible de réaliser des simplifications intermédiaires (projection, suppression-ϵ, déterminisation), qui réduiraient la place mémoire nécessaire et le temps de calcul global.

Quelle que soit l'approche, l'OCR est cependant toujours considéré comme une boîte noire et ne tient pas compte des taux de confiance accordés par le classificateur utilisé. En outre, les ressources nécessaires aux différents systèmes sont relativement importantes, de sorte que leur utilisation sur plateforme embarquée semble difficile.

16.4 Le système de correction proposé

Nous partons de l'hypothèse suivante :

Hypothèse 16.4.1 (Erreurs $n \rightarrow m$)**.** *Les machines à états finis permettent l'expression aisée de règles de gestion des erreurs $n \rightarrow m$ indépendantes du contexte.*

Dans l'ensemble, le système de correction proposé présente les caractéristiques suivantes :

1. Il utilise un dictionnaire et s'inscrit dans l'approche générale décrite au Chapitre 15 : l'accès au dictionnaire est réalisé par une composition filtrée (cf. Section 15.4.2.2), dont le filtre de composition augmente les possibilités d'intersection entre les formes proposées par l'OCR et le dictionnaire. Etant donné un mot w et un dictionnaire $Lexicon$, notre filtre F autorise donc la composition suivante :

   $$w \circ F \circ Lexicon$$

2. Le filtre de composition se distingue fortement des approches de l'état de l'art, étant donné qu'il gère les erreurs de reconnaissance de type $n \rightarrow m$, sans pour autant être dépendant du contexte.
3. Le module de correction s'intègre dans un système de reconnaissance dont le classificateur est un réseau de neurones [3]. Dans ce contexte, notre modèle de correction tire parti des taux de confiance produits par le réseau de neurones : pour chaque caractère, la correction tient compte non pas de la meilleure, mais des 3 meilleures solutions de la classification proposée par l'OCR.
4. La modélisation globale de la correction a été envisagée de manière à ce que le système soit utilisable sur plateforme embarquée : nous avons veillé à limiter les besoins en mémoire vive.

Dans les points suivants, nous détaillons la construction du filtre de composition et les caractéristiques des dictionnaires employés, avant de présenter l'algorithme complet.

16.4.1 Le filtre de composition

Le filtre de composition F que nous proposons se divise en deux modèles distincts : une liste de confusions, $Confus$, et une recherche relâchée, $Match$. La composition filtrée présentée précédemment se réécrit donc comme suit :

$$w \circ (Confus \circ Match) \circ Lexicon$$

16.4.1.1 La liste de confusions

L'objectif de la liste de confusions est de modéliser les erreurs de classification, qu'elles soient dues au système de segmentation ou au réseau de neurones. Il est dès lors

[3] Concernant le principe du réseau de neurones, nous renvoyons à la Section 14.4.6.

nécessaire de déterminer la probabilité qu'une classification soit erronée, et de tenir compte des erreurs d'alignement que la segmentation peut avoir introduites. Formellement,

$$\forall a, b \in \Sigma^+,\ P(b|a) = \frac{\#(a \mapsto b)}{\#(a)} \tag{16.4.1.1}$$

où :

- Σ est l'ensemble des classes gérées par le réseau de neurones.
- a et b sont des classes ou des séquences de classes de Σ.
- $(a \mapsto b)$ signifie « a pour b », « a correspond à b », « a doit être réécrit b ».

Nous proposons donc un modèle capable de gérer les erreurs $1 \rightarrow 1$ ainsi que les erreurs $n \rightarrow m$, et indépendant du contexte.

Entraînement. La liste de confusions que nous utilisons est le résultat d'un entraînement, réalisé sur le corpus de scènes naturelles constitué dans le cadre de la conférence ICDAR'03 (Lucas *et al.* 2003). Les textes de ces images sont en anglais. Les polices de caractères, par contre, ne sont pas dépendantes de la langue. Nous avons de ce fait décidé d'utiliser la liste constituée quelle que soit la langue.

L'entraînement a été réalisé en 4 étapes :

1. Nous avons exécuté le réseau de neurones sur l'ensemble des images du corpus, et récupéré l'étiquetage qu'il propose.
2. Cet étiquetage a été comparé à l'étiquetage de référence. Chaque erreur faite par le réseau de neurones est une confusion qui a été comptabilisée. Etant donné un couple (de séquences) de caractères confondus $\{a, b\}$, nous avons comptabilisé séparément les confusions $(b \mapsto a)$ et $(a \mapsto b)$, de sorte que $P(a|b) \neq P(b|a)$.
3. La liste de confusions a ensuite été élaguée : les nombreuses confusions ne présentant qu'une seule occurrence ont été supprimées, de sorte que la liste contient au total 231 confusions.
4. Nous avons généré un WFST qui modélise les 231 confusions de la liste, mais n'accepte en moyenne qu'une confusion tous les 3 caractères. Cette contrainte permet de réduire le nombre de solutions proposées par le WFST pour une entrée donnée, sans perte au niveau de la qualité de la correction. En effet, les diverses évaluations du réseau de neurones (Thillou *et al.* 2005) ont montré que le taux de reconnaissance sur des scènes naturelles est supérieur à 80%. *A priori*, nous pourrions donc n'accepter une erreur que tous les 5 caractères. Compilé au format binaire, ce WFST prend 2,2 Mo.

Règles de confusion. Le WFST correspondant à la liste de confusions est obtenu par compilation d'un fichier Ovide généré automatiquement à partir de la liste « brute » des confusions relevées lors de la phase d'entraînement. La Figure 16.1 illustre le procédé.

Le fichier Ovide intègre à la fois les confusions apprises et les contraintes sur le nombre de confusions permises en fonction de la longueur. En ceci, ce fichier se distingue des deux fichiers que nous avons utilisés pour modéliser séparément la distance d'édition typographique, où les contraintes sur la longueur et les opérations d'édition autorisées étaient modélisées séparément (cf. Section 15.6.1.2).

Dans la section des classes sont définis des marqueurs de longueur (`L1` à `L4` [4]) et des marqueurs de confusion (`C1` à `C4`). La classe `CHAR` permet simplement de faciliter la lecture : elle correspond aux caractères qui sont reconnus par le réseau de neurones.

Dans la section des règles, les premières règles identifient la longueur des mots à l'aide des marqueurs `L1` à `L4`. Ces marqueurs serviront ensuite de *déclencheurs*, puisque les règles qui réalisent les confusions ne peuvent s'appliquer que si le bon marqueur de longueur est présent. On remarque que chaque règle de confusion insère un marqueur de confusion, `C1` à `C4`. Ce marqueur permet aux règles suivantes de savoir combien de confusions ont déjà été réalisées. L'application d'une règle dépend donc de la longueur identifiée, et du nombre de confusions déjà réalisées.

La première règle de confusion, par exemple, signale que 0 peut devenir 8 quelle que soit la longueur du langage, *pour autant qu'aucun marqueur de confusion n'ait été introduit précédemment*. Si la règle est appliquée, le marqueur de première confusion est inséré. La présence de ce marqueur permet aux règles de deuxième confusion de s'appliquer, pour autant que le langage soit au moins de longueur 2. Le même principe régit les autres niveaux de règles.

Les deux dernières règles suppriment les marqueurs de la machine finale. Ces deux règles pourraient n'en faire qu'une, mais ont été présentées de la sorte pour des questions de lisibilité.

16.4.1.2 La recherche relâchée

Le réseau de neurones ne gère que 36 classes : les 26 lettres de l'alphabet ([a-z]) et les 10 chiffres arabes ($[0-9]$). Ceci signifie que le réseau ne gère ni les différences de casse, ni les accents. Il est dès lors nécessaire de disposer d'un modèle capable de convertir un caractère minuscule non accentué en une majuscule ou un caractère accentué. Par exemple,

- a → [Aäâáàã].
- n → [Nñ].

Le modèle qui réalise ces conversions est un sous-ensemble du modèle de recherche relâchée présenté en Section 15.4.2.2. Dans le cadre du système de reconnaissance de caractères, nous

[4] On constate que le langage `L4` correspond à tous les mots d'au moins 10 caractères.

```
[CLASSIN]
  CHAR     [a-z0-9]
  # longueur du mot
  L1     &1 # 1-3 caracteres
  L2     &2 # 4-6 caracteres
  L3     &3 # 7-9 caracteres
  L4     &4 # 10+ caracteres
  # nombre de confusions
  C1    &5 # 1re confusion
  C2    &6 # 2e confusion
  C3    &7 # 3e confusion
  C4    &8 # 4e confusion
  ...
[RULE]
  # marquer la longueur du mot
  \x00 → <L1> :: ^ _ <CHAR>{1,3} $
  \x00 → <L2> :: ^ _ <CHAR>{4,6} $
  \x00 → <L3> :: ^ _ <CHAR>{7,9} $
  \x00 → <L4> :: ^ _ <CHAR>{10,} $
  # 1re confusion
  0?→ 8 <C1> :: [<L1><L2><L3><L4>] <CHAR>* _ <CHAR>*$ / 6.8579
  m?→ rn <C1> :: [<L1><L2><L3><L4>] <CHAR>* _ <CHAR>*$ / 4.8579
  ...
  # 2e confusion
  0?→ 8 <C2> :: [<L2><L3><L4>] <CHAR>+<C1><CHAR>* _ <CHAR>*$ / 6.8579
  m?→ rn <C2> :: [<L2><L3><L4>] <CHAR>+<C1><CHAR>* _ <CHAR>*$ / 4.8579
  ...
  # 3e confusion
  0?→ 8 <C3> :: [<L3><L4>] [<CHAR><C1><C2>]+ <CHAR>* _ <CHAR>*$ / 6.8579
  m?→ rn <C3> :: [<L3><L4>] [<CHAR><C1><C2>]+ <CHAR>* _ <CHAR>*$ / 4.8579
  ...
  # 4e confusion
  0?→ 8 <C4> :: [<L4>] [<CHAR><C1><C2><C3>]+ <CHAR>* _ <CHAR>*$ / 6.8579
  m?→ rn <C4> :: [<L4>] [<CHAR><C1><C2><C3>]+ <CHAR>* _ <CHAR>*$ / 4.8579
  ...
  # supprimer les marqueurs
  [<L1><L2><L3><L4>] → \x00
  [<C1><C2><C3><C4>] → \x00
```

FIG. 16.1: Fichier Ovide : liste de confusions

```
[CLASSIN]
  A        [Aàâäãáa]
  N        [Nñ]
  ...
[RULE]
  a ?→ <A>
  n ?→ <N>
  ...
```

FIG. 16.2: Fichier Ovide : recherche relâchée en OCR

avons supprimé les conversions superflues (par exemple, A → à) et le coût de la distance d'édition. En effet, au vu de la neutralisation systématique que subissent les caractères, il ne nous a pas paru pertinent de recourir à la distance d'édition, qui défavoriserait inutilement les caractères accentués et les majuscules par rapport aux minuscules non accentuées. La Figure 16.2 illustre le fichier Ovide correspondant. Le FST compilé prend 1,5 Ko.

Notons que la recherche relâchée pourrait être pondérée par un modèle de langue, qui estimerait la probabilité d'une forme a_i particulière au sein d'un ensemble $\overline{a}$ de formes ne différant que par la casse ou l'accentuation :

$$\forall a_i \in \overline{a},\ P(a_i|\overline{a}) = \frac{\#(a_i, \overline{a})}{\#(\overline{a})} \tag{16.4.1.2}$$

Dans ce modèle, la forme minuscule désaccentuée serait bien sûr pondérée également. Il faut en outre ajouter que cette pondération nécessiterait le calcul d'un modèle spécifique à chaque langue traitée.

16.4.2 Le dictionnaire

Contrairement au dictionnaire utilisé en analyse morphologique (cf. Section 15.5.3), le dictionnaire utilisé ici ne contient que les formes lexicales, sans analyse, étant donné que la correction est réalisée en dehors du système de synthèse de la parole. Le dictionnaire est bien entendu dépendant de la langue, et a été compilé avec Ovide, selon la méthode présentée en Section 6.2.

Le dictionnaire anglais que nous employons contient 75 000 formes lexicales, et le dictionnaire français en contient 382 000. Ces chiffres sont trompeurs. En réalité, le dictionnaire anglais couvre 45 000 lemmes, alors que le dictionnaire français n'en couvre que 35 000. Le français est une langue flexionnelle riche, et présente de ce fait plus de formes lexicales pour moins de lemmes. Par contre, le langage correspondant présente de nombreuses redondances, dont peuvent profiter les algorithmes de déterminisation et de minimisation. De ce fait, le FSM du dictionnaire français ne prend que 1,3 Mo, alors que celui de l'anglais prend 2,4 Mo (cf. Table 16.1).

	français	**anglais**
Lemmes	35 000	45 000
Formes fléchies	382 000	75 000
FSM	1,3 Mo	2,4 Mo

TAB. 16.1: Variation de la taille des dictionnaires selon la richesse flexionnelle de la langue

Actuellement, nos lexiques sont de simples FSAs, mais pourraient être pondérés si nous tenions compte de la fréquence des mots dans la langue.

16.4.3 L'algorithme

L'algorithme est présenté en Pseudocode 37. Notre algorithme travaille sur une matrice M, qui contient, pour l'ensemble de la phrase, les 3 meilleures sorties du réseau de neurones. Etant donné un vecteur S qui précise les positions de n espaces dans la chaîne de caractères, l'algorithme commence par segmenter M en $n+1$ mots.

Comme le montre la Figure 16.3, pour chaque mot, un WFSA w est construit dynamiquement et mémorisé dans le vecteur W, un vecteur de pointeurs de WFSAs (ligne 1). En intégrant les 3 meilleures sorties de l'OCR pour chaque lettre, w constitue une manière simple et élégante de représenter l'ensemble des suites de lettres possibles.

Dans w, chaque sortie de l'OCR se voit attribuer un poids, en tenant compte du comportement de l'OCR :

1. La sortie la mieux classée reçoit fréquemment plus de 90% du taux de confiance, tandis que les autres sorties se partagent les 10% restants.
2. Les deuxième et troisième sorties sont généralement mieux pondérées que les suivantes, mais sont très souvent difficiles à distinguer : leur différence de pondération n'est pas *significative*.
3. Utiliser le taux de confiance de l'OCR ne produit pas un système viable, parce que la première solution y est trop avantagée. Or, si le système nécessite un module de correction, c'est parce qu'il arrive au reconnaisseur de se tromper...

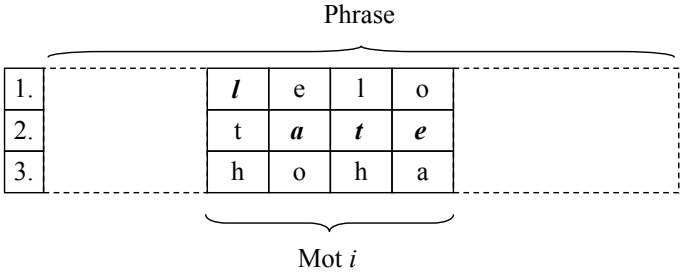
De ce fait, nous avons expérimentalement accordé un poids 3 fois plus important à la première sortie qu'aux deux autres :

$$-\log_2 P(out[i]) = \begin{cases} -\log_2(0,6) & \text{si } i = 1 \\ -\log_2(0,2) & \text{sinon} \end{cases} \tag{16.4.3.1}$$

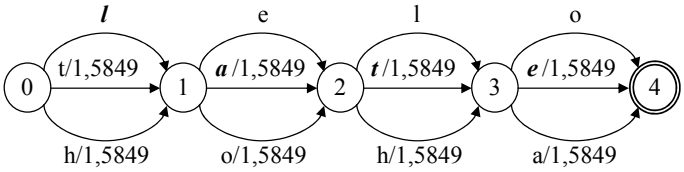
où $out[i]$ est la i^e sortie du réseau de neurones, $1 \leq i \leq 3$. Pour rappel, les poids sont exprimés sous la forme de logarithmes, parce que nos machines à états finis sont définies sur le semi-anneau tropical.

$W \leftarrow$ `FSMVecCreate`(M, S, n)
$B \leftarrow$ `new FSM`$[n{+}1]$
for $i = 0$ to n **do**
  $B[i] \leftarrow \emptyset$
  $\rho \leftarrow W[i]$
  for $j = 0$ to m **and** $\rho \neq \emptyset$ **do**
    $\rho \leftarrow$ `FSM_Compose`$(\rho, \mathcal{M}[j])$
    `FSM_Determinize`(`FSM_EpsilonFree`(`FSM_Project`$(\rho,$ `OUT`$)$)) **if** $\rho \neq \emptyset$
  end for
  if $\rho \neq \emptyset$ **then**
    $B[i] \leftarrow$ `FSM_GetBestPath`(ρ)
  end if
  if $B[i] = \emptyset$ **or** $\omega(B[i]) > \theta(W[i])$ **then**
    $B[i] \leftarrow$ `FSM_GetBestPath`$(W[i])$
  end if
end for
return `StringCreate`(B)

Pseudocode 37: modèle de correction orthographique en scènes naturelles



M, matrice des 3 meilleures sorties de l'OCR



w, WFSA du mot i

W, vecteur de $n{+}1$ pointeurs de WFSA

FIG. 16.3: Depuis M, la matrice OCR, à W, le vecteur de pointeurs de WFSA

L'algorithme itère ensuite sur les machines mémorisées dans W (lignes 3–15). Chaque WFSA $W[i]$ est combiné, par composition, avec m (W)FSTs, qui représentent les différents modèles ($\mathcal{M}$) que nous utilisons (lignes 6–9). Actuellement, les modèles sont donc, dans l'ordre : la liste de confusions, la recherche relâchée et le dictionnaire. Après chaque composition avec le modèle courant, le résultat ρ subit trois simplifications (ligne 8) : la sortie est projetée, pour supprimer les variations présentes en entrée et devenues inutiles ; la suppression-ϵ est ensuite appliquée, étant donné que la projection a pu révéler des transitions vides ; enfin, la machine est déterminisée, parce que le résultat de la suppression-ϵ est une machine non déterministe.

A la fin de cette itération, si ρ n'est pas nul, le WFSA $B[i]$ est construit (ligne 11) : il contient le meilleur chemin de ρ, c'est-à-dire la forme de notre dictionnaire qui est la plus proche du mot i, étant donné les modifications que nos modèles intermédiaires autorisent. Ce meilleur chemin a un poids, $\omega(B[i])$, combinant le poids attribué à chaque caractère par le réseau de neurones, et les poids attribués à chaque opération d'édition par les modèles de correction.

Cependant, si $B[i]$ n'a pas été construit, ou si $\omega(B[i])$ est supérieur à un seuil θ calculé sur $W[i]$, $B[i]$ est remplacé par le meilleur chemin de $W[i]$ (ligne 14). Ceci permet de gérer de manière très efficace les OOVs. Expérimentalement, le seuil a été défini comme suit :

$$\theta(W[i]) = -\log_2(0,25)\,|w_i| \tag{16.4.3.2}$$

où $|w_i|$ est la longueur du mot i. Nous acceptons donc, en moyenne, tout candidat dont le poids ne dépasse pas 0,25 par caractère. Ce seuil a évidemment été déterminé en fonction des probabilités de confusion du réseau de neurones. Dans le principe, il interdit au système de correction d'accepter plusieurs corrections majeures, mais permet d'accepter plusieurs corrections mineures.

L'algorithme se termine en construisant, à partir du vecteur B, la string correspondant à la phrase complète (ligne 17). La phrase peut maintenant être présentée au système de synthèse.

16.5 Evaluation

16.5.0.1 Taux de correction

Nous proposons deux évaluations : la première a été réalisée sur un corpus français et la seconde, sur un corpus anglais. Dans les deux cas, nous comparons les résultats obtenus aux résultats d'autres méthodes de correction. Pour faciliter la lecture, notre système de correction basé sur les machines à états finis et mettant en œuvre un filtre de composition est appelé *correction FSM*.

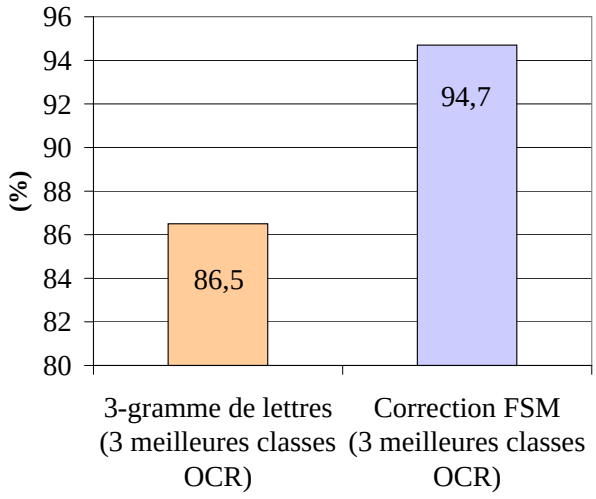


FIG. 16.4: Evaluation sur notre corpus français

Evaluation en français. Cette évaluation a été réalisée sur un corpus de 400 scènes naturelles, construit dans le cadre du développement de l'application de lecture embarquée décrite au début de ce chapitre. Ces scènes naturelles sont réalistes : elles ont été acquises par des personnes aveugles, en conditions réelles. Elles présentent de ce fait l'ensemble des caractéristiques que nous avons mentionnées précédemment : mots tronqués, problèmes de contraste et de luminosité, etc.

Notre correction FSM a été comparée au premier modèle de correction intégré au système, utilisant un tri-gramme de lettres (Thillou *et al.* 2005). La Figure 16.4 montre une amélioration significative du taux de correction, puisque la correction FSM corrige 8,2% de mots de plus que le tri-gramme initial.

Ce bon résultat, obtenu sur un corpus français, montre en outre que la liste de confusions n'est pas spécifique au corpus d'entraînement, qui était un corpus anglais : notre liste de confusions modélise bien les erreurs de l'OCR, et non le corpus d'apprentissage.

Evaluation en anglais. Pour cette évaluation, nous avons employé le corpus ICDAR'03 lui-même, qui inclut 2 268 scènes naturelles. La correction FSM est ici comparée au système commercial *ABBYY FineReader 8.0 Professional Edition Try&Buy*.

L'évaluation a été préparée comme suit. Après une phase de segmentation adaptée aux scènes naturelles (Mancas-Thillou 2006), les images ont été soumises au système commercial, qui a atteint un taux de reconnaissance de 70,9%. Des 2 268 scènes du corpus, 660 (29,1%) ont donc été mal reconnues.

L'évaluation se focalise exclusivement sur ces erreurs, et compare les 3 meilleures *suggestions* du système commercial à *la solution* retenue par notre correction FSM. En outre, deux configurations de la correction FSM ont été évaluées : dans la première, le module de

correction n'utilise que la meilleure solution du classificateur, tandis que dans la seconde, il bénéficie des 3 premières solutions du classificateur. Le Figure 16.5 présente les résultats obtenus.

Le premier constat est que la correction FSM, quelle que soit la configuration retenue, propose un taux de corrections nettement supérieur au taux de suggestions du système commercial. Le second constat est que le recours aux 3 meilleures solutions du classificateur améliore considérablement nos propres performances, mais au prix d'une augmentation simultanée du nombre de fausses alarmes : certains mots corrects, mais absents de notre dictionnaire, sont modifiés à tort par le système. Notons que la modification ne donne pas toujours un mot du dictionnaire. Par exemple, *Constantine*, qui est hors-vocabulaire, est modifié en *Comstantine* parce que le seuil de correction accepté a été dépassé, et que la meilleure solution du réseau de neurones présente un m et non un n.

Ne pas considérer le module de reconnaissance comme une boîte noire semble donc une approche pertinente, mais qui demande probablement une amélioration de la prise en compte des résultats du classificateur, et un perfectionnement des modèles linguistiques utilisés.

Notre correcteur gère efficacement les erreurs $n \rightarrow m$. Le mot *Office* (Figure 16.6, gauche), par exemple, est corrigé par la correction FSM alors que le classificateur l'avait reconnu comme *Ohice*. Le système commercial, quant à lui, n'a pu proposer le mot correct : les suggestions qu'il fait se limitent à la distance de Levenshtein classique.

Le correcteur gère également correctement certains problèmes de cadrage. Le mot *form* (Figure 16.6, droite), par exemple, est un cas typique de caractères coupés dans une scène naturelle. Notre classificateur le reconnaît comme *fnrm*, mais la seconde proposition du réseau de neurones pour la lettre n est o, ce qui permet à la correction FSM de trouver la bonne solution, et de classer *form* devant *farm*. On voit ici tout l'intérêt d'intégrer, dans la correction, plusieurs solutions du classificateur.

16.5.0.2 Temps de traitement

Le temps de traitement de notre module de correction a été estimé en conditions réelles, sur un PDA de configuration standard : processeur Intel cadencé à 520 MHz, 64 Mo SDRAM [5], Windows Mobile 2003. Le corpus sur lequel a été estimé le temps de traitement est celui de notre premier test : 400 scènes naturelles de mots français.

En moyenne, le module de correction traite un mot de 13 caractères en 0,2 s. Au vu des limites de la plateforme considérée, ces performances sont tout à fait acceptables.

Le temps de calcul et le fait que le système fonctionne sur PDA sont une conséquence directe des simplifications (projection, suppression-ϵ, déterminisation) réalisées après chaque composition entre le résultat courant et le modèle courant. Sans ces simplifications, le système ne tient tout simplement plus dans les 64 Mo de mémoire disponibles...

[5] SDRAM : *Synchronous Dynamic Random Access Memory*.

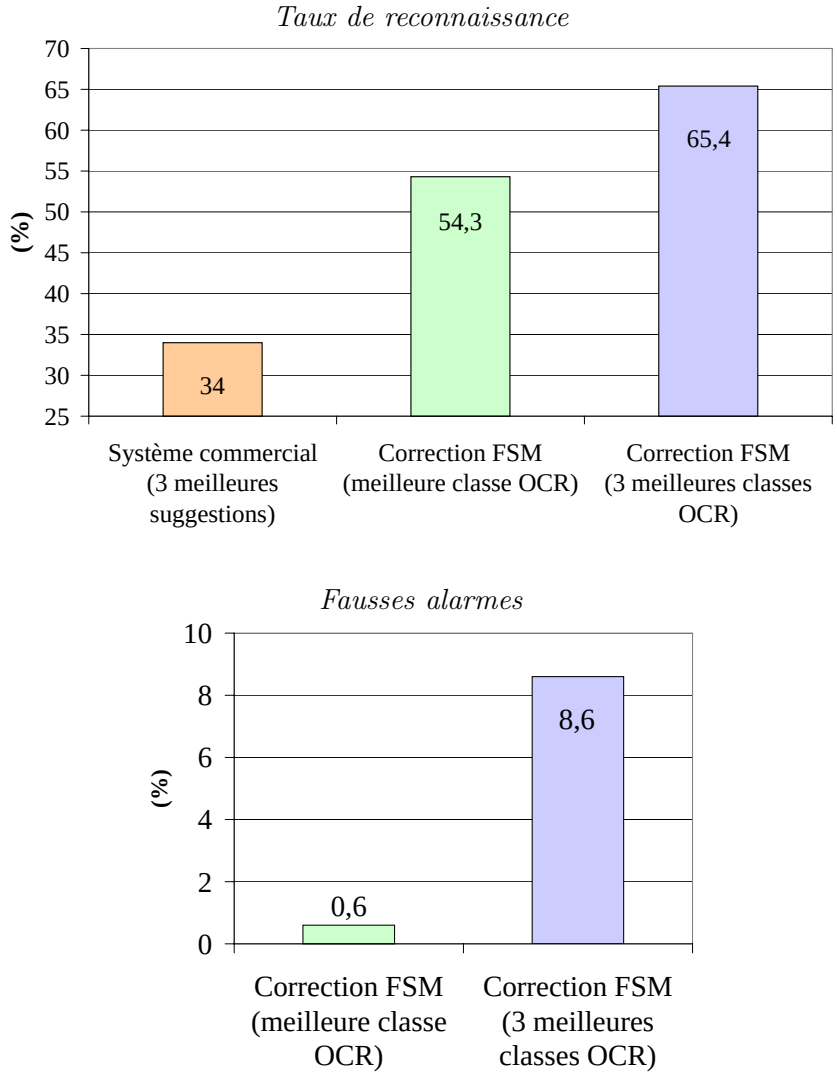


FIG. 16.5: Evaluation sur le corpus ICDAR'03

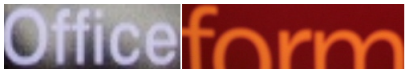


FIG. 16.6: Deux scènes naturelles difficiles, gérées par la correction FSM

16.6 Synthèse

16.6.1 Une approche efficace

L'approche proposée combine, de manière élégante, les 3 meilleures sorties d'un réseau de neurones à un filtre de composition, qui réalise une distance de Levenshtein complexe entre la forme à reconnaître et les mots d'un dictionnaire. Dans l'ensemble, on constate que cette approche pertinente est facilitée par le recours aux machines à états finis, qui permettent d'intégrer de manière simple des niveaux de représentation fort différents.

Au sein des machines à états finis, les 3 meilleures solutions du réseau de neurones sont représentées sous la forme d'un treillis de solutions. Contrairement aux approches classiques, il n'est donc pas nécessaire de générer l'ensemble des strings correspondant aux diverses combinaisons possibles.

La caractéristique la plus importante de cette approche est certainement sa capacité à modéliser une distance de Levenshtein complexe : contrairement à la distance classique, qui gère des erreurs comme *rn* pour *m*, ou *h* pour *li*, à l'aide de plusieurs opérations d'édition, notre modèle a la capacité de considérer ces erreurs comme des substitutions $n \rightarrow m$. Ceci permet de favoriser le *bon alignement* parmi l'ensemble des alignements possibles. La méthode a en outre l'avantage de modéliser ces substitutions $n \rightarrow m$ indépendamment de tout contexte, ce qui permet leur application à n'importe quelle string, qu'elle ait été ou non observée sur le corpus d'entraînement.

L'approche FSM a en outre l'avantage de favoriser un calcul rapide de la distance entre la sortie de l'OCR et l'ensemble des mots du dictionnaire : la représentation du dictionnaire sous la forme d'un automate minimisé permet de naturellement factoriser l'ensemble des comparaisons communes aux strings qui partagent les mêmes préfixes et les mêmes suffixes. Ce mode de représentation permet donc de gérer des dictionnaires contenant plusieurs milliers de formes. Le dictionnaire du français, par exemple, compte plus de 380 000 formes...

L'algorithme proposé tire également avantage des opérations définies sur les machines à états finis. Nous ne construisons pas un seul transducteur. Les différents modèles sont composés en cours de processus, ce qui permet de réduire la taille des résultats intermédiaires, en leur appliquant les opérations définies sur les machines à états finis : la projection, la suppression-ϵ et la déterminisation. Ce sont ces opérations qui permettent de limiter la place mémoire nécessaire, dans des proportions telles que le module peut être utilisé sur plateforme embarquée, malgré le peu de place mémoire disponible dans ce type d'environnement.

16.6.2 Des limites à dépasser

L'approche proposée est évidemment encore fort limitée. Comme nous l'avons signalé au début de ce chapitre, nous avons volontairement limité l'approche aux seuls OOVs.

La correction FSM profitera par exemple certainement de l'analyse morpho-syntaxique d'eLite, lorsque notre système de synthèse sera disponible sur plateforme embarquée.

Cependant, la correction FSM des OOVs, indépendamment de la fusion avec eLite, pourrait intégrer d'autres modèles d'analyse. Une piste que nous comptons suivre prochainement est celle de la segmentation morphologique ou syllabique. L'objectif de l'approche serait de valider les constituants – morphèmes ou syllabes – d'une forme inconnue de notre dictionnaire. Ceci permettrait de résoudre deux problèmes non résolus actuellement. Le premier est la correction abusive d'une forme qui serait inconnue, mais qui présenterait une cohérence syllabique ou morphémique. Le second est l'absence de correction, alors que la forme le nécessite ; c'est par exemple le cas de la forme *Comstantine*, où *Coms-* pourrait paraître suspect, et accepter la correction *Cons-*, phonétiquement similaire.

Un autre modèle absent est la gestion des espaces. Notons à ce propos que nous avons abordé la question, et construit un modèle *Space* gérant l'insertion et la suppression d'espaces dans l'ensemble d'une scène naturelle. Dans cette optique, l'algorithme est évidemment différent, puisqu'il traite en une seule étape l'ensemble des mots S de la scène :

$$S \circ (Confus \circ Match \circ Space) \circ Lexicon$$

Cependant, ce modèle augmente significativement le temps de traitement sur PDA, et rend le système inutilisable en pratique. Quelques tests indiquent sans ambiguïté que le modèle d'espaces autorise un trop grand nombre de solutions *valides*. Une autre approche doit donc être tentée. Notre hypothèse actuelle est que l'information la plus pertinente, pour mettre en doute la segmentation proposée, se situe au niveau du module de segmentation lui-même, qui prend la décision de segmentation en fonction d'un taux de confiance donné. Ce taux de confiance pourrait être intégré au modèle de correction. Le tout est de déterminer la manière la plus efficace de le faire...

Ainsi, de nombreuses améliorations peuvent encore être apportées au modèle proposé, qui n'a donc certainement pas atteint les limites de ses possibilités.

Chapitre 17

Conclusion

Cette partie a été entièrement consacrée à la correction orthographique dédiée à la synthèse de la parole.

17.1 Positionnement du problème

Parce que les erreurs les plus fréquentes auxquelles est confronté un système de synthèse sont celles commises par l'utilisateur lorsqu'il entre le texte au clavier, nous nous sommes principalement concentré sur la modélisation d'un système de correction capable de gérer ces erreurs. Il arrive cependant que le texte à prononcer provienne d'une étape de reconnaissance de caractères, susceptible de corrompre le texte. Nous avons dès lors proposé un second modèle, dédié aux erreurs dues au système de reconnaissance.

Quelles que soient les erreurs à gérer, l'approche proposée devait tenir compte des deux caractéristiques qui font la particularité de la synthèse de la parole. D'une part, le confort de l'auditeur peut être gêné par les erreurs si elles sont audibles dans le flux de parole. La correction en synthèse doit donc traiter ces erreurs particulières en priorité. D'autre part, la seule interaction autorisée entre la synthèse et le ou les utilisateurs est le flux de parole généré ; l'utilisateur ne peut intervenir dans le processus de correction. Nous sommes donc face à la nécessité d'une correction déterministe.

17.2 Objectifs

Dans l'ensemble, nous avons poursuivi deux objectifs. D'une part, nous voulions montrer que les machines à états finis permettent de modéliser de nombreuses tâches complexes comme une succession d'étapes simples. D'autre part, nous voulions montrer que la composition filtrée, basée sur la notion de distance d'édition, permet de gérer de manière similaire des cas de figure en apparence fort différents.

17.3 Etat de l'art

17.3.1 Correction des OOVs

L'état de l'art a montré que les recherches en correction des OOVs sont fort avancées : de nombreuses méthodes ont été proposées. Leur analyse a révélé les caractéristiques suivantes :

1. Toutes les méthodes proposées produisent des résultats très satisfaisants. Le choix de l'une d'entre elles est probablement plus orienté par des considérations subjectives ou par les compétences du développeur, que par une différence fondamentale dans les résultats obtenus.
2. De l'ensemble des méthodes existantes, seule la distance d'édition a fait l'objet d'une modélisation à l'aide de machines à états finis. Ces modélisations, très efficaces en terme de temps de traitement, n'ont cependant pas été pensées pour s'intégrer dans un système géré exclusivement à l'aide de machines à états finis.
3. Le seul véritable point faible de la correction des OOVs est la modélisation des erreurs $n \rightarrow m$, dont nous avons vu l'importance en reconnaissance des caractères. Le seul modèle proposé est fortement dépendant de la qualité du corpus d'apprentissage, parce que ces erreurs sont gérées en tenant compte des contextes rencontrés dans le corpus.

17.3.2 Correction des IVs

La correction des IVs a également suscité un vif intérêt de la part des chercheurs. De nombreuses approches linguistiques ont été proposées. Ces approches, qui ont été les premières dans le domaine, sont certainement efficaces et pertinentes, si ce n'est qu'elles rendent difficile la mise au point d'un système multilingue. Notre système de synthèse se veut multilingue, ce qui limite l'intérêt d'une approche purement linguistique.

Pour dépasser cet inconvénient, les chercheurs se sont tournés vers les méthodes statistiques. Les premiers systèmes proposés se sont basés sur des modèles de langue à orientation lexicale. Cependant, la méthode a buté sur la difficulté à proposer une estimation consistante de l'ensemble des suites lexicales possibles. Ceci a conduit la majorité des chercheurs à limiter leurs travaux aux seules listes de confusion. De très nombreuses méthodes ont été proposées dans ce domaine particulier de la correction dépendante du contexte, et produisent des résultats très satisfaisants. Cependant, les listes de confusion gèrent exclusivement les éléments de la liste. Cette approche n'est donc pas générique, ce qui limite considérablement son intérêt en synthèse, où la variété des textes à traiter impose une approche plus englobante.

Actuellement, la recherche en correction orthographique s'intéresse davantage à la gestion des erreurs survenant dans les requêtes réalisées sur Internet, *via* un moteur de recherche. Les méthodes comparatives, qui exploitent les requêtes similaires précédemment réalisées, s'éloignent considérablement des enjeux qui sont les nôtres.

A notre connaissance donc, aucune approche, en correction des IVs, ne s'est intéressée à ce qui constitue les spécificités de la synthèse de la parole. Nous avons cependant retenu de l'un des modèles proposés qu'une approche stratifiée, corrigeant progressivement les erreurs présentes dans une forme, pouvait faciliter la modélisation d'un système de correction gérant à la fois les OOVs et les IVs.

17.4 Correction des textes entrés au clavier

La mise en place d'une correction des textes entrés au clavier ne s'est pas limitée à la définition de modèles de correction. L'analyse des faiblesses de l'état de l'art nous a en effet conduit à poser les postulats suivants :

Postulat 15.1.2 (Approche globale). *La correction orthographique doit faire partie intégrante du processus d'analyse linguistique, de manière à profiter au maximum de l'ensemble des informations disponibles. Le processus de correction peut donc proposer des solutions en cours de traitement, mais doit attendre la fin du traitement avant de prendre une décision.*

Postulat 15.1.1 (Erreurs audibles). *En synthèse, un système de correction doit se concentrer sur les erreurs d'orthographe qui sont décelables dans le flux de la parole, parce qu'elles en rendent l'écoute inconfortable.*

Postulat 15.1.3 (Gestion des treillis). *Etant donné une erreur, les solutions trouvées par une étape de correction peuvent en proposer un découpage différent. Il devient dès lors difficile de les mémoriser dans la structure de données du synthétiseur, dans l'attente d'un choix ultérieur.*

Postulat 15.1.4 (Limitation des transferts de données). *Tout traitement doit éviter de stocker dans la structure de données des informations qui seront modifiées ou supprimées par les traitements suivants. Respecter cette contrainte limite le temps imparti à la conversion des données entre traitement et structure de stockage.*

Postulat 15.6.1 (Limites d'édition). *Le nombre d'erreurs autorisées dans une forme lexicale doit tenir compte de trois facteurs : la longueur de la forme, le type de l'erreur et les limites de la compréhension humaine.*

Sur la base de ces postulats, nous avons décidé de reconsidérer la totalité de l'analyse linguistique d'eLite, afin que la correction profite effectivement de son intégration dans le processus d'analyse.

17.4.1 Nouvelle analyse morpho-syntaxique

Nous avons complètement revu l'architecture de l'analyse telle qu'elle se présentait initialement : dans la première version de l'analyse, un pré-processeur, un analyseur morphologique et un analyseur syntaxique traitaient séquentiellement le texte et communiquaient au travers d'une structure de données, la DLS. Afin de conserver cette approche, nous aurions dû considérablement modifier la DLS, afin qu'elle accepte de nouvelles informations, comme les caractéristiques flexionnelles, et qu'elle autorise la représentation de treillis de solutions proposées par la correction.

Nous avons dès lors considéré qu'il était plus utile de fusionner les analyses morphologique et syntaxique en un seul module, de manière à ce que la sauvegarde des données dans la DLS puisse se limiter au strict nécessaire.

Cette nouvelle configuration nous a offert la possibilité de modéliser la totalité de l'analyse morpho-syntaxique à l'aide de machines à états finis. Elle a cependant nécessité de nombreuses adaptations, afin de conserver l'intégrité de l'analyse initiale :

1. Afin de gérer les niveaux *Word* et *Unité linguistique* de la DLS, une interface de communication a dû être définie. Selon cette interface, chaque processus veille à ce que la machine qu'il construit présente le niveau *Word* en entrée, et le niveau *Unité linguistique* en sortie. Dans le principe, cette représentation est obtenue par l'application d'opérations comme la composition et la projection, définies sur les transducteurs.
2. Nous avons dû définir des méthodes adaptées, permettant de repérer, dans une machine, un type particulier de données et de segmenter la machine en fonction de cette information. C'est ainsi qu'ont été définies les opérations de *test dichotomique* et de *segmentation*, qui sous-tendent la totalité de l'algorithme proposé.
3. Le modèle de langue a dû être réparti entre l'analyse morphologique et l'analyse syntaxique. Le modèle lexical a été confié à l'analyse morphologique, l'analyse syntaxique se limitant dès lors au modèle syntaxique. Dans le principe, le modèle complet est dès lors reconstitué par composition de deux machines qui réalisent chacune une partie du modèle global.

Cette nouvelle architecture intègre de nouveaux modèles :

1. La recherche relâchée, déjà présente dans l'ancienne analyse, a été améliorée sous la forme d'une composition filtrée pondérée, utilisée lors de tout test dichotomique.
2. Un modèle de casse a été ajouté, afin de gérer efficacement les noms propres et les acronymes.
3. Une analyse morphologique spécifique aux OOVs a été définie, et réalise une analyse pondérée des flexions. Cette analyse n'est cependant nécessaire que si le modèle de casse et les modèles de correction des OOVs n'ont donné aucun résultat.
4. Un modèle syntagmatique fort simple, qui augmente le modèle syntaxique classique. Il permet la gestion de certains syntagmes qui dépassent la fenêtre d'analyse du *n*-

gramme, mais restent relativement réduits. Nous avons principalement modélisé des groupes verbaux.

Ces nouveaux modèles participent à l'amélioration de l'analyse, dont les résultats sont 3% supérieurs à ceux de l'analyse précédente. Dans cette nouvelle architecture, la correction orthographique peut maintenant profiter des résultats de l'analyse réalisée.

17.4.2 La correction

Nous avons montré que deux types d'erreurs sont certainement audibles : celles qui résultent en de véritables non-mots, qui sont audibles quel que soit le contexte, et les erreurs d'accord, pour autant que le contexte implique une liaison. Les modèles proposés concernent donc ces deux types d'erreurs.

17.4.2.1 Correction des non-mots

Dans le processus d'analyse, un non-mot ne subit cette étape de correction que lorsque le modèle de casse n'a donné aucun résultat.

En considérant que le texte analysé a été entré au clavier, les non-mots du texte sont le résultat de deux causes. La première est la proximité des touches sur le clavier, qui entraîne des erreurs typographiques. La seconde est la méconnaissance de l'orthographe d'usage, qui entraîne certaines approximations phonétiques. Dans les deux cas, nous avons proposé des modèles inspirés de la notion de distance d'édition.

Lorsque la distance typographique se limite à une différence de casse ou d'accentuation, la recherche relâchée suffit à la correction. Dans le cas contraire, la distance typographique que nous avons proposée tient compte de la disposition des touches sur le clavier afin de limiter les opérations d'édition autorisées. Elle tient en outre compte de la longueur du mot, afin de déterminer le nombre d'opérations d'édition permises. Ces trois caractéristiques distinguent notre modèle de l'état de l'art.

La distance phonétique proposée a été établie à partir d'un dictionnaire de phonétisation. Elle détermine l'ensemble des graphèmes homophones, mais n'autorise la réécriture d'un graphème que si le graphème proposé en remplacement a été rencontré dans le bon contexte. Le contexte utilisé est limité à une lettre de part et d'autre du graphème. Notre modèle se distingue de l'état de l'art par cette contextualisation des transformations autorisées, et par l'absence de tout calcul phonétique en cours de traitement.

Dans l'algorithme d'analyse, un non-mot se voit attribuer une liste non élaguée de candidats. Le choix d'une correction n'est cependant réalisé qu'à la fin de l'analyse syntaxique, lors du choix du meilleur chemin dans le treillis de possibilités. Dans l'ensemble, les taux de correction obtenus démontrent la pertinence de l'approche.

Notre modèle de correction souffre cependant de l'absence d'un modèle capable de gérer les abréviations et les mots volontairement tronqués. L'approche est en outre indéniablement limitée par la qualité du dictionnaire utilisé.

17.4.2.2 Correction des erreurs d'accord

Telle que nous l'avons envisagée, la correction des erreurs d'accord implique de connaître l'analyse flexionnelle des formes de la phrase. Cette étape de la correction n'est dès lors réalisée qu'après l'analyse syntaxique, lorsqu'une correction a été attribuée aux non-mots de la phrase.

Le modèle de correction comporte trois étapes : une étape de détection des erreurs potentielles, une étape de génération des flexions aux endroits signalés par la détection, et une étape d'analyse flexionnelle, qui choisit, à l'aide d'un modèle de langue, la meilleure suite de flexions dans un treillis de possibilités.

Cette approche est la seule du genre. Elle est générique, parce qu'elle ne se limite pas à une liste de confusions. Elle se distingue des approches linguistiques, parce qu'elle ne remplace pas une flexion suspecte, mais propose une ou plusieurs alternatives. Il s'agit en outre de la première tentative de correction flexionnelle à l'aide d'un modèle de langue.

Sur notre corpus de test, les résultats sont appréciables. Ils montrent en tout cas que notre approche stratifiée, où les non-mots sont corrigés avant les erreurs flexionnelles, est pertinente.

Cependant, nous émettons une réserve majeure vis-à-vis de ce modèle. Les étapes de détection et d'analyse flexionnelle n'ont qu'une vision locale de la phrase. Par manque de contexte, le système ne constate pas certaines erreurs, et réalise certaines corrections abusives. Du modèle proposé, nous retenons donc principalement l'efficacité du système de génération.

Le système bénéficierait certainement d'une analyse syntagmatique. Il ne semble pas impossible de la concevoir à l'aide de machines à états finis, bien que ceci demande mure réflexion. La véritable question, nous semble-t-il, est plutôt de savoir si l'approche est, dans ce cas, encore suffisamment générique, pour rester multilingue...

17.5 Correction des textes extraits de scènes naturelles

L'application dans laquelle nous avons intégré un modèle de correction orthographique nous a imposé deux contraintes. D'une part, le système complet devait être portable sur plateforme embarquée. D'autre part, la correction ne pouvait être intégrée au système de synthèse, qui n'était pas le nôtre. Nous nous sommes de ce fait concentré exclusivement sur la correction des OOVs.

L'état de l'art dans le domaine particulier de la reconnaissance de caractères en scènes naturelles, a mis en évidence deux faiblesses des modèles proposés. D'une part, le système de reconnaissance est toujours considéré comme une boîte noire, sans considération pour les taux de confiance qu'il peut accorder à ses décisions. D'autre part, les erreurs de reconnaissance résultent fréquemment en des erreurs de type $n \rightarrow m$, qui sont généralement traitées comme des séquences de plusieurs opérations d'édition de base. Un système cependant gère ces erreurs, mais au prix d'une contextualisation qui diminue la généricité

du procédé proposé. De cette seconde faiblesse, nous avons dégagé le postulat suivant :

Postulat 16.2.1 (Erreurs $n \rightarrow m$). *Une gestion des erreurs $n \rightarrow m$, indépendante du contexte, est une nécessité en reconnaissance des caractères.*

Outre les erreurs classiques $1 \rightarrow 1$, le système que nous avons proposé gère les erreurs $n \rightarrow m$ sans aucune dépendance au contexte. Les probabilités de confusions sont directement apprises à partir des erreurs de classification du classificateur utilisé. La représentation de ces erreurs est incontestablement facilitée par l'utilisation de machines à états finis. Les évaluations réalisées montrent que les confusions apprises sont indépendantes du corpus d'apprentissage, et améliorent considérablement les résultats, sans générer pour autant de fausses alarmes.

Notre modèle prend en outre en compte les trois meilleures solutions du système de reconnaissance. Les évaluations réalisées ont montré tout l'intérêt de cette information, qui augmente considérablement le nombre des corrections, au prix cependant de quelques modifications abusives.

Le système présenté est très efficace, mais encore rudimentaire. L'analyse des résultats nous incite à envisager deux pistes d'amélioration. D'une part, il serait probablement intéressant de tenir compte de la consistance syllabique ou morphémique des formes inconnues du lexique, afin d'éviter les modifications abusives. D'autre part, le système devrait intégrer un mécanisme de gestion des espaces. Pour ce faire, le module de segmentation du système de reconnaissance devrait probablement participer au processus de correction.

17.6 Les apports des machines à états finis

17.6.1 Au niveau de l'analyse morpho-syntaxique

La totalité de l'analyse est représentée sous la forme de machines à états finis, principalement parce que le format de représentation nécessaire peut être obtenu par application de certaines opérations, comme la composition et la projection. Dans l'ensemble, les opérations définies sur les machines à états finis autorisent donc la *gestion de treillis de solutions*. Sans ces opérations, les treillis devraient constamment être parcourus et convertis dans la structure de données commune à l'ensemble de nos modules d'analyse.

De nouvelles opérations, le test dichotomique et la segmentation, facilitent la mise en œuvre de notre Hypothèse 1, qui pose l'intérêt de *diviser pour mieux régner*.

17.6.2 Au niveau des modèles de correction

Tous les modèles de correction présentés reposent sur la notion fondamentale de *composition filtrée*. Ils sont en outre décrits à l'aide de règles de réécriture qui exploitent les *marqueurs* que nous avons définis.

Composition filtrée. La composition filtrée est une extension de la distance d'édition des automates proposée par Mohri. La distance d'édition est donc calculée au travers d'un filtre de composition. Ceci présente les avantages suivants :

1. Le filtre est un transducteur pondéré. Il peut donc être le résultat de la compilation de règles de réécriture pondérées. De ce fait, les relations les plus complexes peuvent être modélisées. Ceci facilite, par exemple, la modélisation du clavier de la distance typographique, la contextualisation de la distance phonétique ou l'expression des erreurs $n \rightarrow m$ en reconnaissance des caractères.
2. Le filtre peut être le résultat de la composition de plusieurs machines à états finis. Ceci permet de modéliser une relation complexe, difficilement concevable par l'esprit humain, sous la forme de plusieurs sous-relations beaucoup plus simples, à la portée de l'esprit humain. Le principe permet donc d'éviter les erreurs de modélisation. La distance typographique en est un bon exemple, puisqu'elle correspond à la composition de la recherche relâchée, des contraintes sur la longueur et de la modélisation du clavier. La distance d'édition de la reconnaissance en est un autre, puisque le modèle combine les 3 meilleures sorties du classificateur, la liste de confusions et la recherche relâchée.
3. Un filtre complexe peut être conservé sous la forme de ses constituants, composés en cours de traitement. Les résultats intermédiaires sont de ce fait simplifiables, ce qui limite la place mémoire nécessaire ainsi que le temps de traitement. Le meilleur exemple de ceci est certainement le système de correction utilisé en reconnaissance des caractères, sur plateforme embarquée : sans simplifications intermédiaires, l'application n'est pas envisageable.

Marqueurs. L'utilisation de marqueurs présente incontestablement les deux avantages suivants :

1. Ils facilitent l'écriture des règles, en réduisant la nécessité de gérer des contextes complexes. Dans le principe, une règle insère un marqueur en fonction d'un contexte. Toutes les règles dont l'application nécessite ce contexte peuvent dès lors se contenter de préciser le marqueur, qui sert dès lors de déclencheur. Les règles sont plus faciles à écrire, mais également plus rapides à compiler.
2. Ils évitent les erreurs de modélisation. Les marqueurs concernés ici sont les masqueurs, qui évitent à un symbole réécrit par une règle de devenir la cible d'autres règles suivantes, et les bloqueurs, qui empêchent que la suppression de symboles n'entraîne la constitution d'un contexte favorable à l'application d'autres règles.

Les modèles de correction proposés ne sont donc pas simplement représentés sous la forme de machines à états finis. Ils sont concevables, *parce que* nous employons des machines à états finis.

17.7 Des hypothèses confirmées

L'analyse des résultats obtenus avec les deux modèles de correction proposés démontre le bien-fondé des hypothèses suivantes :

Hypothèse 15.2.1 (Gestion des treillis). *Les machines à états finis permettent de conserver les solutions d'un traitement sous la forme d'un treillis où les divergences de découpage importent peu.*

Hypothèse 15.2.2 (Limitation des transferts de données). *Les machines à états finis évitent les transferts inutiles entre le traitement et le stockage, en conservant l'ensemble des informations dans un format directement utilisable d'un traitement à l'autre.*

Hypothèse 15.2.3 (Approche stratifiée). *La correction orthographique peut utilement s'organiser en couches, de manière à corriger progressivement les formes qui contiennent différents types d'erreurs.*

Hypothèse 15.2.4 (Casse et accentuation). *Il est fréquent qu'une forme lexicale ne soit pas reconnue par l'analyse alors qu'elle ne diffère de la forme recensée dans le lexique qu'au niveau de la casse ou de l'accentuation. Il est dès lors important de détecter ces formes au plus tôt, afin de leur éviter un traitement lourd à réserver aux véritables erreurs typographiques.*

Hypothèse 15.5.1 (Dictionnaire d'IVs). *En synthèse, les formes lexicales connues du système ne nécessitent pas d'analyse morphologique. Un dictionnaire suffit amplement, pour autant que la structure représentant le dictionnaire soit légère et autorise un accès rapide. Les machines à états finis respectent ces contraintes.*

Hypothèse 15.5.2 (Analyse morphologique des OOVs). *En synthèse, les formes lexicales inconnues sont les seules formes à nécessiter une véritable analyse morphologique. Cette analyse est nécessaire afin d'orienter au mieux le choix de la catégorie à attribuer à la forme lors de l'analyse syntaxique.*

Hypothèse 15.6.1 (Limites d'édition). *Un filtre de composition est le média idéal pour modéliser aisément les trois facteurs qui influent sur le nombre d'erreurs présentes dans une forme lexicale.*

Hypothèse 16.4.1 (Erreurs $n \rightarrow m$). *Les machines à états finis permettent l'expression aisée de règles de gestion des erreurs $n \rightarrow m$ indépendantes du contexte.*

Les Hypothèses 15.2.3, 15.2.4, 15.5.1 et 15.5.2 s'inscrivent par ailleurs dans la lignée de notre hypothèse initiale :

Hypothèse 1 (Diviser pour mieux régner). *Pour autant qu'une tâche complexe puisse être analysée comme une succession d'étapes simples, les machines à états finis constituent l'outil idéal pour représenter cette succession d'étapes simples de manière aisée.*

Enfin, les données dépendantes de la langue sont complètement externalisées. La seule limite, peut-être, de l'analyse morpho-syntaxique présentée est qu'elle dépend encore du comportement général des langues flexionnelles. Le système de correction proposé en scènes naturelles est par contre complètement indépendant de la langue. Tel qu'il a été conçu, il peut d'ailleurs accepter l'ajout de nouveaux modèles sans être recompilé, puisque les modèles utilisés sont composés au sein d'une procédure itérative. Notre seconde hypothèse initiale semble donc confirmée :

Hypothèse 2 (Externalisation des données). *Pour autant que les langues traitées partagent suffisamment de similarités, les machines à états finis autorisent l'externalisation de l'ensemble des traitements dépendant de la langue. Tout l'art réside dès lors dans la détection, au sein d'un processus, des traitements qui ressortissent à la langue.*

17.8 Une hypothèse non confirmée...

L'hypothèse suivante n'est malheureusement pas confirmée par notre évaluation :

Hypothèse 15.6.2 (Modèle de langue flexionnel). *Les performances des modèles de langue en analyse syntaxique laissent penser qu'un modèle de langue, combinant les catégories syntaxiques et les traits grammaticaux, peut efficacement gérer les erreurs d'accord présentes dans une phrase.*

17.9 Un accueil encourageant et révélateur

Lors de sa présentation à ICDAR'07, notre modèle dédié aux scènes naturelles a rencontré un intérêt tout particulier, auquel nous ne nous attendions pas. Les chercheurs du domaine nous ont fait remarquer qu'ils s'étaient posé de nombreuses questions quant à l'utilisation de machines à états finis en post-correction de la reconnaissance de caractères, et que notre système leur montrait l'intérêt de cette technologie.

Ces mêmes chercheurs ont en outre attiré notre attention sur un point peut-être évident, mais auquel nous n'avions pas songé : les machines à états finis sont certainement utiles, mais elles ne sont utilisables que si l'on possède un compilateur permettant l'expression des langages et des relations désirés. La construction manuelle de la plupart des machines décrites dans cette partie est en effet totalement inconcevable...

Conclusion

1 Du contexte aux objectifs

Les machines à états finis, représentables sous la forme de graphes orientés étiquetés, sont les équivalents informatiques des langages réguliers et des relations régulières. Elles acceptent de ce fait les opérations régulières, qui facilitent la construction de langages et de relations complexes à partir de langages et de relations simples. Elles s'accompagnent en outre d'algorithmes d'optimisation, qui permettent d'une part de les représenter en un minimum de place, et d'autre part de conserver un temps de parcours linéairement proportionnel à la taille d'une entrée à analyser. Elles peuvent enfin être exprimées sous la forme d'expressions régulières et de règles de réécriture, descriptions syntaxiques qui en facilitent considérablement la modélisation. Les machines à états finis sont donc des outils puissants, certainement utiles au traitement du langage naturel.

Ces outils, en l'état, restent cependant limités. Les langages réguliers, par nature incapables de gérer les imbrications infinies, ne sont pas particulièrement adaptés à la description syntaxique des langues. En outre, de nombreuses applications en traitement du langage naturel nécessitent la possibilité d'exprimer un certain degré d'incertitude, ce que les langages réguliers non pondérés n'autorisent pas. Les chercheurs en langage naturel, dans leur grande majorité, se sont de ce fait désintéressés de ces outils.

La recherche a cependant récemment avancé de manière considérable dans le domaine des machines à états finis pondérées : dans le contexte des séries rationnelles, les algorithmes initialement dédiés aux machines non pondérées ont été étendus aux machines pondérées, et de nouveaux algorithmes, comme la recherche des n meilleurs chemins d'un graphe, ont été spécialement définis pour les machines pondérées. Ce véritable bond en avant de la recherche a ouvert la voie à une nouvelle technologie du langage, basée sur des machines pondérées et capable de modéliser cette incertitude, indispensable à certains domaines du traitement du langage naturel.

La synthèse de la parole en est un bon exemple : le processus de synthèse de la parole, qui consiste à produire de la parole à partir du texte, regorge de difficultés, empreintes d'un certain degré d'incertitude, qui peuvent utilement profiter du potentiel expressif des outils à états finis dans leur ensemble. Deux de ces difficultés ont particulièrement retenu notre

attention :

1. *La sélection des unités.* Dans le contexte de la synthèse par sélection d'unités non uniformes, l'étape de sélection des unités est délicate. Il n'est pas évident de proposer un système de sélection capable de combiner efficacement les bons critères de sélection et la bonne méthode d'optimisation.
2. *La correction orthographique.* Les recherches ne se sont pas encore intéressées aux contraintes toutes particulières de la correction orthographique en synthèse de la parole : la synthèse est particulièrement sensible aux fautes d'orthographe audibles, et se doit d'intégrer une solution déterministe, puisque les interactions entre le système et l'utilisateur se limitent à la parole générée.

C'est ainsi que s'est progressivement dessiné l'objectif général de cette thèse : nous avons voulu proposer, en synthèse de la parole, des solutions aux problèmes particuliers de la sélection d'unités non uniformes et de la correction orthographique, développées exclusivement sur la base de machines à états finis. Cet objectif, fort général, recouvre en réalité trois objectifs de nature fort différente :

1. En tout premier lieu, nous voulions prouver d'une part que les machines à états finis peuvent, à elles seules, modéliser des tâches complexes, et d'autre part qu'elles permettent de modéliser ces tâches complexes comme une succession d'étapes simples.
2. Ce faisant, nous désirions proposer des approches nouvelles et originales dans nos deux domaines d'application.
3. Dans une moindre mesure, nous avions l'espoir de contribuer au domaine des machines à états finis, par la proposition de nouveaux algorithmes ou de nouvelles méthodes d'implémentation.

2 Quelques contributions aux machines à états finis

Ces contributions s'inscrivent dans le cadre du développement de nos propres outils : une nouvelle bibliothèque de machines à états finis et un compilateur d'expressions régulières et de règles de réécriture pondérées.

2.1 Extension des modes de représentation

Notre bibliothèque a été implémentée selon les principes de la programmation orientée objet. Elle met en œuvre l'encapsulation, le polymorphisme et l'héritage, de sorte que l'utilisateur manipule l'ensemble des machines représentables au travers d'une seule et même interface.

Deux de nos contributions, principalement d'ordre informatique, s'inscrivent résolument dans cette perspective, et participent au polymorphisme de la bibliothèque. Il s'agit

de deux nouveaux modes de représentation des machines, permettant de réduire considérablement la place nécessaire à leur représentation, lorsque les langages modélisés respectent certains critères :

1. *Les classes de symboles.* Notre bibliothèque autorise la définition de classes de symboles, qui condensent en une seule transition l'ensemble des transitions, de même poids, qui relient deux états d'une machine. Cette originalité est pertinente lorsque le langage modélisé est de la forme $\Sigma^* \alpha$.
2. *Les graphes orientés pondérés.* Les machines peuvent être représentées sous la forme de graphes orientés pondérés, qui font l'économie des symboles qui étiquettent classiquement les transitions entre états. Ce mode de représentation s'applique exclusivement aux langages pensés et conçus comme des graphes, dont les états correspondent aux symboles de l'alphabet.

2.2 Extension des règles de réécriture

Notre compilateur permet la description de règles de réécriture, de langages et de dictionnaires pondérés. Dans le principe, les règles de réécriture semblent relativement faciles à concevoir. Construire un ensemble de règles, exempt d'erreurs de modélisation, est cependant un exercice délicat.

Notre troisième contribution se situe dans ce contexte. Afin de faciliter l'expression de contraintes et d'éviter les erreurs de modélisation, nous avons défini la notion de *marqueur*, qui permet d'identifier un phénomène et d'en suivre l'évolution. Le marqueur peut être :

1. Un *déclencheur*, qui indique de manière non ambiguë qu'une condition d'application a été rencontrée.
2. Un *masqueur*, qui évite l'application erronée d'une règle sur une expression régulière.
3. Un *bloqueur*, qui empêche la formation d'une expression régulière sur laquelle une règle pourrait s'appliquer à tort.

La combinaison des marqueurs et du mécanisme d'inclusion de fichiers de notre compilateur facilite fortement l'expression de modèles complexes. Cette contribution s'inscrit donc résolument dans le domaine de la formalisation des langages.

3 Les originalités des approches proposées

3.1 Sélection d'unités non uniformes

Postulats. La méthode de sélection d'unités proposée se fonde sur l'analyse des systèmes de l'état de l'art. De cette analyse, nous avons dégagé un certains nombre de postulats et d'hypothèses. Les postulats posés sont les suivants :

1. (Postulat 10.1.1) Nous avons considéré que l'unité linguistique idéale en synthèse est le diphone, parce qu'il assure une concaténation sur la partie stable du signal.

2. (Postulat 10.1.2) Nous avons en outre choisi de conserver l'accent de mot.
3. (Postulat 10.2.1.) Nous avons estimé que la structure de représentation ne peut modifier l'algorithme de sélection de manière involontaire, mais peut par contre ajouter des contraintes qui améliorent l'algorithme.

Caractéristiques. Sur cette base, nous avons construit un système dont les caractéristiques sont les suivantes :

1. (Hypothèse 10.1.1) Nous avons proposé un système de sélection qui fait une distinction stricte entre les caractéristiques linguistiques, réservées au coût cible, et les caractéristiques acoustiques, réservées au coût de concaténation.
2. (Hypothèses 10.1.2 et 10.1.3) Considérant que les caractéristiques linguistiques participent à une prise de décision prosodique, nous avons proposé de les pondérer par entropie. Afin de tenir compte des différences articulatoires entre phonèmes, nous avons réalisé la pondération par phonème.
3. (Hypothèse 10.2.2) L'unité de sélection étant le diphone, nous avons considéré que les informations segmentales pouvaient être supprimées, ainsi que de nombreuses informations suprasegmentales, impliquées par la structure du diphone.
4. (Hypothèse 10.2.3) Nous fondant sur des études linguistiques et des expérimentations, nous avons accordé une importance certaine à la pause et à la structure syllabique dans la détermination de la longueur syllabique.
5. (Hypothèse 10.2.1) Les machines à états finis nous ont permis d'optimiser le processus de sélection tout en respectant l'algorithme initial.
6. (Hypothèse 10.2.4) Le recours aux machines à états finis a en outre permis d'exprimer des contraintes sur le graphe de concaténation, qu'il aurait été difficile d'exprimer au travers d'un autre formalisme.

Algorithme. L'algorithme proposé se divise en quatre étapes principales, réalisées exclusivement à l'aide de machines à états finis, dans lesquelles toutes les listes de candidats et tous les coûts nécessaires sont pré-calculés. Les opérations majeures, dans l'ensemble du processus, sont la composition, la concaténation et la projection. Les trois premières étapes s'appliquent séparément aux cibles d'une phrase, tandis que la quatrième concerne la phrase complète.

La première étape vérifie que le diphone n'est pas un diphone manquant, et le remplace par un ou plusieurs diphones présents le cas échéant. La seconde étape construit la cible à partir du diphone et de ses caractéristiques. S'il s'agit d'une cible manquante ou d'une cible présente, mais en trop petit nombre, la cible est remplacée ou complétée par des cibles présentes. La troisième étape convertit les cibles en identifiants numériques. Lorsque toutes les cibles de la phrase ont été traitées, la quatrième étape construit un treillis d'identifiants, et le repondère à l'aide du graphe de concaténation. La meilleure suite d'identifiants est finalement sélectionnée dans ce treillis.

Analyse. Les évaluations réalisées confirment le bien-fondé de l'approche proposée. Dans l'ensemble, le système de sélection permet l'obtention d'une parole de qualité, en un temps de traitement considérablement réduit.

3.2 Correction orthographique

3.2.1 Correction des textes entrés au clavier

La majeure partie du travail réalisé en correction orthographique a concerné la correction des textes entrés au clavier, étant donné que ce périphérique est le média d'entrée de texte le plus employé.

Postulats. Le système de correction orthographique proposé part des postulats suivants :

1. (Postulat 15.1.1) Les erreurs à traiter en synthèse sont les erreurs audibles. Sur cette base, nous avons décidé de nous concentrer sur les erreurs qui résultent en des non-mots, ainsi que sur les erreurs d'accord. Les erreurs sémantiques et les erreurs de style n'ont pas été considérées.
2. (Postulat 15.1.2) Nous avons en outre estimé que le processus de correction devait s'inscrire dans le processus d'analyse morpho-syntaxique, de manière à intégrer, dans la prise de décision, l'ensemble des informations disponibles.
3. (Postulats 15.1.3 et 15.1.4) Nous avons enfin considéré qu'il était nécessaire de limiter les transferts de données entre les traitements et la structure de données, d'une part parce que certains traitements génèrent des treillis de solutions difficiles à parcourir, et d'autre part parce qu'il est nécessaire d'éviter de perdre du temps à de simples transferts d'information.
4. (Postulat 15.6.1) Le nombre d'erreurs autorisées dans une forme lexicale doit dépendre de la forme, et non des limites du système.

Caractéristiques. L'établissement de l'état de l'art en correction orthographique nous a convaincu que de nombreuses méthodes pertinentes existaient dans le domaine de la correction des non-mots. Nous avons par contre constaté que seuls les systèmes experts avaient abordé la question de la correction des erreurs d'accord, mais ne proposaient pas d'approche globale et multilingue. Sur la base de cet état de l'art et des postulats que nous avons posés, nous avons élaboré un système de correction dont les caractéristiques sont les suivantes :

1. (Hypothèses 15.2.1 et 15.2.2) Afin de faciliter la gestion des treillis et la limite des transferts de données, nous avons fusionné les deux modules d'analyse morpho-syntaxique, et nous avons proposé de réaliser l'intégralité des traitements à l'aide de machines à états finis.
2. (Hypothèses 15.5.1 et 15.5.2) Nous avons estimé que l'analyse morphologique *stricto sensu* devait être réservée aux mots hors-vocabulaire non corrigés par le système de

correction. Les mots du lexique, quant à eux, peuvent se contenter d'un accès à un dictionnaire. Ceci est fortement facilité par l'utilisation de machines à états finis.

3. (Hypothèse 15.2.4) Nous avons proposé une gestion de la casse et de l'accentuation indépendante de la correction des non-mots, de manière à détecter ce type d'erreurs au plus tôt et à leur éviter un traitement lourd et inutile.
4. (Hypothèse 15.6.1) Nous nous démarquons de l'état de l'art en proposant une correction des non-mots dépendante de la forme. Nous proposons une distance d'édition qui tient compte de la longueur de la forme, du type de l'erreur et des limites de la compréhension humaine. Deux distances sont prises en compte : une distance typographique, et une distance cognitive ou phonétique.
5. (Hypothèse 15.2.3) Nous avons estimé que la correction peut utilement s'organiser en couche, afin de corriger progressivement les formes qui contiennent différents types d'erreurs.
6. (Hypothèse 15.6.2) Etant donné les bons résultats des modèles de langue en analyse syntaxique, et étant donné notre objectif de rester multilingue, nous avons considéré que les modèles de langue étaient suffisants pour fonder un système efficace de gestion des erreurs d'accord.

Algorithme. Les divers traitements réalisés communiquent *au travers* de machines à états finis, ce qui a nécessité de nouvelles méthodes de gestion des machines à états finis, permettant la réalisation de tests dichotomiques et la segmentation des machines en fonction de ces tests. Cette architecture a également nécessité de répartir le modèle de langue entre l'analyse morphologique et l'analyse syntaxique. Outre les méthodes spécifiquement développées, les opérations majeures, dans l'ensemble du processus, sont la composition, la concaténation et la projection.

La totalité de l'algorithme repose sur la mise en œuvre d'une recherche relâchée, qui permet de déterminer au plus tôt le traitement approprié à appliquer à une forme. L'algorithme réalise un traitement morphologique en fonction de l'unité linguistique traitée. Le véritable lieu de la correction des non-mots est l'analyse des formes lexicales. Une forme lexicale ne subit cependant la correction destinée aux non-mots que si la recherche relâchée a été insuffisante. Les mots connus des lexiques sont pondérés lors de l'accès au dictionnaire, tandis que les mots véritablement inconnus sont pondérés par une analyse morphologique des terminaisons.

Lorsque la totalité d'une phrase a été traitée, elle est analysée par un modèle de langue, qui ne conserve que la meilleure séquence du treillis disponible. Cette séquence est ensuite soumise à la correction flexionnelle. Le processus de correction comprend trois étapes. La première étape détecte les erreurs potentielles et propose des alternatives en termes de traits linguistiques. La seconde étape génère les flexions correspondant aux alternatives. La troisième étape est un modèle de langue flexionnel, qui choisit le meilleur chemin dans ce

treillis de possibilités flexionnelles. Ce chemin est sauvegardé dans la structure de données du système.

Analyse. Les évaluations réalisées valident certainement l'approche dans son ensemble, qui est efficace et suffisamment rapide.

Les résultats obtenus au niveau de la correction des non-mots mettent en évidence la pertinence des modèles proposés, qui s'adaptent à la forme concernée. Ils indiquent également qu'un système de gestion complet des non-mots doit posséder un dictionnaire de qualité, et devrait disposer d'une méthode de gestion des abréviations et des mots tronqués.

L'évaluation de la correction flexionnelle semble également positive. Les erreurs commises par le système montrent cependant que certaines limites ne pourront être dépassées à l'aide de modèles de langue. La correction flexionnelle devrait plutôt recourir à une analyse syntaxique. Ce point est abordé dans nos perspectives (cf. Section 5). Nous émettons par contre quelques doutes quant à la possibilité d'une approche multilingue, dans ces conditions.

L'approche proposée a certainement le mérite de recourir exclusivement à des machines à états finis, qui facilitent de nombreux traitements. Il serait donc intéressant de conserver le principe, fût-il nécessaire de remplacer certains modèles.

3.2.2 Correction en scènes naturelles

Nous avons également eu l'occasion de proposer un modèle de correction des non-mots dans le contexte d'une application embarquée, basée sur un système de reconnaissance de caractères.

Postulats. Ce modèle se fonde sur le postulat (Postulat 16.2.1) qu'une gestion des erreurs $n \rightarrow m$, indépendante du contexte, est une nécessité en reconnaissance des caractères. Dans l'état de l'art, aucune méthode convaincante ne tient compte de ce type d'erreurs. Les systèmes n'intègrent en outre jamais le taux de confiance du classificateur dans la prise de décision de la correction.

Caractéristiques. Le système que nous avons proposé a les caractéristiques suivantes :

1. (Hypothèse 16.4.1.) Nous avons entraîné une liste de confusions $n \rightarrow m$ sur les erreurs du réseau de neurones intégré au système de reconnaissance. Cette liste de confusions a été modélisée aisément sous la forme d'une machine à états finis.
2. Nous avons modélisé séparément l'absence d'accentuation, étant donné que le reconnaisseur les supprime systématiquement.
3. Nous avons intégré les trois premières solutions du réseau de neurones dans la prise de décision.

Algorithme. L'algorithme se divise en deux étapes. Les opérations majeures sont la composition, la concaténation et la projection.

Une première étape construit un treillis de solutions pondérées à partir de la sortie du réseau de neurones. Une seconde étape itère sur les différents modèles proposés, dont le dernier est le dictionnaire de la langue.

Le meilleur chemin du treillis de solutions est ensuite sélectionné, pour autant qu'un seuil donné n'incite pas le système à préférer la première solution du reconnaisseur.

Analyse. L'évaluation du système montre la pertinence de l'approche proposée. Les erreurs de reconnaissance sont très bien gérées, et les véritables mots inconnus sont bien respectés par l'approche dans son ensemble.

Nous avons en outre constaté à quel point la composition des modèles en cours de traitement est importante : elle autorise des simplifications intermédiaires, sans lesquelles l'application ne pourrait tourner sur plateforme embarquée.

4 Les apports des machines à états finis

De manière générale, nous constatons que toutes les approches proposées recourent aux machines pondérées et ne pourraient s'en passer. La gestion d'un certain degré d'incertitude est donc effectivement une condition *sine qua non* à l'utilisation des machines à états finis dans nos domaines d'application.

Les machines pondérées n'auraient cependant pas une grande utilité, si l'opération de *composition* n'avait pas été définie. Cette opération est la base des principaux apports relevés dans le cadre de cette thèse :

1. La composition constitue le fondement de la compilation des règles de réécriture, qui simplifient considérablement la conception de relations régulières.
2. La composition facilite la conception de relations complexes. Les modèles linguistiques que nous avons proposés sont, dans la plupart des cas, le fruit de la composition de plusieurs modèles simples, représentés initialement par des ensembles de règles de réécriture distincts.
3. La composition est la base de la notion de *composition filtrée*, qui généralise la distance d'édition de deux automates. Des distances d'édition complexes peuvent de ce fait être obtenues, par compilation de plusieurs ensembles de règles de réécriture. Ceci est totalement inconcevable en dehors des outils à états finis.
4. La composition permet de diviser un processus en une succession d'étapes simples. C'est sur cette base qu'ont été conçus tous nos algorithmes. Ceci offre plusieurs avantages de taille :
   (a) La division d'un processus autorise des simplifications intermédiaires. Dans ce contexte, la projection et la déterminisation complètent utilement la composition.

Sans ces simplifications, certaines applications ne seraient pas concevables. C'est le cas, par exemple, de la correction sur plateforme embarquée.

(b) Les filtres de composition peuvent être conservés sous la forme de leurs constituants. Ceci permet de recalculer une partie sans devoir recalculer le tout. La phase de modélisation en est facilitée.

(c) Un modèle peut être dispersé en plusieurs endroits d'un processus. C'est le cas de notre modèle de langue, reconstitué progressivement par composition.

Nous constatons donc que cette opération, à elle seule, permet de confirmer notre Hypothèse 1, selon laquelle les machines à états finis facilitent la mise en œuvre de notre Postulat 1, établissant l'intérêt de diviser pour mieux régner.

Nous posions dans la même section le Postulat 2, établissant la nécessité d'une séparation stricte entre l'algorithme et les données, dans la conception d'un système multilingue. Les développements réalisés dans cette thèse mettent en évidence que les machines à états finis facilitent la mise en œuvre de ce postulat, et confirment ainsi notre Hypothèse 2.

Notre modèle de correction en scènes naturelles, testé en français et en anglais sans aucune modification de l'algorithme, en est un bon exemple. Notre modèle de sélection d'unités non uniformes en est un autre : le système n'est en rien dépendant des caractéristiques retenues en français, et acceptera, sans aucune modification de l'algorithme, que d'autres langues aient recours à d'autres caractéristiques, que ce soit au niveau du coût cible ou du coût de concaténation.

La seule contrainte à respecter, bien sûr, est que les différentes machines, employées pour une langue donnée, respectent les mêmes conventions. En quelque sorte, on leur demande de… parler *le même langage*.

Incontestablement, les machines à états finis sont donc des outils puissants, dont l'exploitation permet la conception d'applications légères, efficaces et performantes.

5 Perspectives

Certains des modèles que nous avons proposés en correction orthographique, s'ils sont intéressants, ont leurs limites :

1. Les résultats obtenus en correction flexionnelle indiquent sans ambiguïté que les modèles de langue ne permettent pas une correction contextuelle de qualité. Une véritable analyse syntaxique semble dans ce cas plus pertinente.

2. La gestion des mots hors-vocabulaire est encore élémentaire, parce qu'elle est incapable de gérer les mots corrects, mais absents du dictionnaire.

5.1 Analyse syntaxique

5.1.1 Point de départ

Les modèles de langue, qui ont montré toute leur pertinence en désambiguïsation syntaxique (cf. Section 13.5.2.2), sont par contre la source de nombreuses erreurs en correction flexionnelle : la correction flexionnelle implique de modifier la phrase, et dans ce contexte, le modèle de langue devient incontrôlable. En somme, la vision réduite que le modèle de langue a de la phrase n'est pas gênante en analyse, mais devient un véritable handicap en *génération*.

Nous en concluons que la correction flexionnelle nécessite le soutien d'une véritable analyse syntaxique, capable de construire la structure arborescente de la phrase. Cette structure, qui a été présentée en Section 13.5.2.1, devrait permettre au système de correction de prendre connaissance des dépendances syntaxiques existant au sein de la phrase à corriger.

La représentation arborescente de la structure syntaxique d'une phrase est le résultat de l'application des règles de dérivation d'une grammaire formelle (cf. Section 1.3), qui organise les catégories syntaxiques d'une phrase en syntagmes de différents niveaux, jusqu'à remonter à l'axiome de la grammaire, souvent noté S. Les catégories syntaxiques sont les symboles terminaux de la grammaire, tandis que les syntagmes et l'axiome en sont les symboles non-terminaux.

Quelle grammaire formelle ? Parmi les grammaires formelles figurent les grammaires régulières, équivalentes aux langages réguliers. Contrairement aux grammaires de plus haut niveau, comme les grammaires contextuelles et hors-contexte, les grammaires régulières sont incapables de gérer les imbrications infinies. Une grammaire régulière rendant compte de la syntaxe d'une langue donnée n'est cependant pas inconcevable : en effet, s'il est vrai que la langue autorise les imbrications infinies, il est également certain que les productions linguistiques ne les présentent jamais, parce que les locuteurs ne sont pas capables de les appréhender. Cependant, comme le constate Chomsky (1956), les grammaires régulières manquent d'expressivité. Une grammaire régulière, qui décrirait la syntaxe d'une langue, manquerait donc de concision et serait difficile à concevoir. Il est dès lors préférable de recourir à des grammaires de plus haut niveau.

A priori, la description des phénomènes syntaxiques d'une langue nécessite le recours à une grammaire contextuelle, où la réécriture des non-terminaux peut être contextualisée. Or, les langages produits par ces grammaires sont ceux reconnus par machines de Turing linéairement bornées non déterministes. Une telle grammaire est donc inutilisable en pratique, parce que le problème de l'appartenance d'une séquence de symboles au langage d'une grammaire dépendante du contexte est *pspace*-complet.

Les chercheurs ont eu, de ce fait, recours aux grammaires hors-contexte. Ces grammaires sont pourtant plus adaptées à la description des langages de programmation qu'à la description des langues : un langage de programmation autorise les imbrications infinies

et sa syntaxe est, par nature, non-contextuelle. Malgré ce manque d'expressivité, le choix des grammaires hors-contexte est dû au fait que ces grammaires produisent des langages reconnus par automates à pile non-déterministes. Les grammaires hors-contexte sont donc utilisables en pratique [1]. En somme, le choix de la grammaire a été dirigé par des considérations d'utilisabilité, au détriment de l'expressivité.

Ainsi, le choix de décrire la syntaxe d'une langue à l'aide de grammaires hors-contexte est un compromis entre expressivité et utilisabilité. Le compromis est certainement nécessaire. Par contre, les choix qui ont été réalisés ne nous conviennent pas :

1. S'il est indispensable de recourir à une grammaire de plus haut niveau que la grammaire régulière, il semble que son choix ne doive pas dépendre de son utilisabilité, mais de ses capacités expressives. *A priori*, il faut donc recourir aux grammaires contextuelles.
2. L'efficacité des machines à états finis est, en outre, un atout considérable qu'il convient de conserver.

Partant, nous en concluons que l'idéal est de profiter du pouvoir expressif des grammaires contextuelles, mais de les convertir en grammaires régulières, de manière à pouvoir les représenter sous la forme de machines à états finis. Une méthode de *conversion régulière* d'une grammaire de plus haut niveau est donc nécessaire.

5.1.2 Aperçu de l'état de l'art en conversion régulière

Dans l'établissement de ces perspectives, nous n'avons pas la prétention de réaliser un état de l'art complet du domaine. Nous invitons de ce fait le lecteur intéressé à se reporter à (Nederhof 2000a), qui en a constitué une bibliographie extensive. Nous réalisons ci-dessous une analyse succincte des éléments qui caractérisent l'ensemble des approches définies. Nous tâchons de nous positionner par la même occasion.

Les différentes méthodes proposées dans la littérature présentent 3 points communs : le type de grammaire considérée, l'approximation du langage initial et l'absence d'arbre syntaxique.

Grammaire considérée. Toutes les méthodes concernent les grammaires hors-contexte. Or, Nous l'avons signalé, il paraît intéressant de partir d'une grammaire contextuelle, qui permet une meilleure description des phénomènes linguistiques.

Approximation. Deux faits ont été démontrés. Premièrement, la question de savoir si une grammaire hors-contexte génère un langage régulier est indécidable (Harrison 1978). Deuxièmement, convertir une grammaire hors-contexte, qui génère un langage régulier, en un automate à états finis acceptant le même langage est un problème insoluble (Ullian 1967). Aucune méthode ne peut donc garantir que le langage est préservé lorsque la grammaire

[1] Notons que l'automate à pile non-déterministe n'assure pas que le parcours d'une séquence de symboles reste linéairement proportionnel à la longueur de la séquence.

génère déjà un langage régulier. Au mieux, il y a approximation de la grammaire. Selon la méthode, cependant, la grammaire régulière obtenue accepte soit un sous-ensemble (Pulman 1986, Johnson 1998a), soit un super-ensemble (Pereira & Wright 1997, Nederhof 2000b, Mohri & Nederhof 2001) du langage initial.

Super-ensemble. Construire un langage régulier qui est un super-ensemble du langage initial ne présente pas d'intérêt dans notre cas. En effet, ceci signifie que des séquences de terminaux, initialement hors-langage, sont acceptées par la grammaire après approximation.

C'est le cas, par exemple, lorsque Nederhof (2000b) transforme la grammaire hors-contexte, qui accepte le langage $\{a^n\, b^n \mid n \geq 0\}$, en la grammaire régulière, qui accepte le langage $\{a^*\, b^*\}$: les séquences $\{a^n\, b^m \mid n, m \geq 0 \text{ et } n \neq m\}$, qui étaient initialement rejetées, sont dès lors acceptées. La grammaire devient donc permissive. Sans entrer ici dans le détail, ceci est dû au fait que l'algorithme d'approximation, en interdisant les imbrications infinies, supprime dans le même temps le parenthésage de la grammaire initiale.

Sous-ensemble. Les approches qui réduisent le langage accepté répondent certainement mieux aux besoins de la correction flexionnelle, pour autant que la réduction reste dans des limites raisonnables et compréhensibles.

Parmi les méthodes réductrices, certaines, comme celle de Pereira & Wright (1997), commencent par construire un automate à pile, qui est ensuite converti en automate à états finis. Comme le constatent Mohri & Nederhof (2001), la différence de structure entre l'automate à pile et la grammaire hors-contexte est telle que l'auteur de la grammaire hors-contexte ne peut prédire la forme de la grammaire régulière générée, ni influencer cette dernière. Ce type d'approche n'est donc pas à conseiller.

Bien que la méthode qu'il décrit construise un super-ensemble du langage initial, Nederhof (2000b) présente un principe très intéressant qui permet de conserver, dans la grammaire régulière, le parenthésage initial de la grammaire hors-contexte pour une profondeur donnée. Le principe est fort simple : les symboles non-terminaux récursifs de la grammaire initiale sont remplacés, dans la nouvelle grammaire, par de nouveaux non-terminaux non-récursifs, que l'on ajoute à l'alphabet des non-terminaux de la grammaire. Par exemple, le parenthésage de la grammaire

$$S \quad \rightarrow \quad aSa \mid bSb \mid \epsilon$$

peut être conservé sur une profondeur d. Dans l'exemple suivant, d=3 :

$$\begin{array}{lcl} S[1] & \rightarrow & aS[2]a \mid bS[2]b \mid \epsilon \\ S[2] & \rightarrow & aS[3]a \mid bS[3]b \mid \epsilon \\ S[3] & \rightarrow & aSa \mid bSb \mid \epsilon \\ S & \rightarrow & aSa \mid bSb \mid \epsilon \end{array}$$

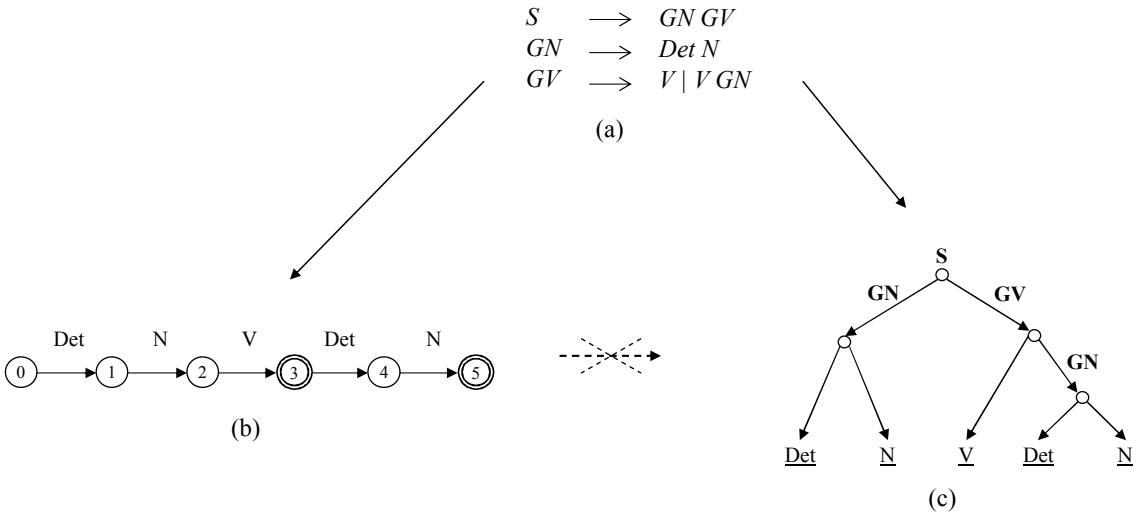


FIG. 18.1: Grammaire régulière simpliste (a) et l'automate correspondant (b). L'arbre syntaxique (c), correspondant à la séquence « *Det N V Det N* », ne peut être obtenu à partir de l'automate

La dernière règle de la nouvelle grammaire est toujours récursive, et sera traitée, dans un deuxième temps, par l'algorithme d'approximation régulière, qui crée un super-ensemble de la grammaire initiale à partir de la profondeur d+1. Nous retenons de cette approche le principe qui permet de conserver le parenthésage initial sur d niveaux.

arbre syntaxique. Quelle que soit la méthode, l'automate à états finis, généré à partir de la grammaire régulière, ne gère que les terminaux de la grammaire décrite. Ceci est illustré par la Figure 18.1 : la grammaire régulière simpliste (a) correspond à l'automate (b), qui permet d'accepter la séquence « *Det N V Det N* », mais sans pouvoir construire l'arbre syntaxique (c) correspondant.

Or, comme nos tests l'ont montré, la structure syntaxique est une information indispensable à l'analyse flexionnelle. Nous ne pouvons dès lors nous limiter aux méthodes de conversion d'une grammaire régulière en un automate à états finis proposées dans la littérature.

5.1.3 Ebauche d'une approche complète

Dans cette section, nous proposons une nouvelle approche. Nous attirons l'attention sur le fait que l'approche proposée en est encore à l'état d'ébauche : elle n'a fait l'objet d'aucune démonstration, n'a été testée que sur quelques grammaires relativement simples et présente encore de nombreuses limitations.

Notations. Une grammaire formelle donnée est notée G. L'approximation régulière correspondante est notée G'. Le langage accepté par une grammaire G (resp. G') est noté $L(G)$ (resp. $L(G')$), et la machine à états finis acceptant ce langage est notée $M(G)$ (resp. $M(G')$). Une phrase est notée P, son automate, $A(P)$, et son arbre syntaxique, $Tree(P)$.

Principes généraux. La méthode que nous proposons se fonde sur les principes suivants :

1. Etant donné que $L(G')$ sera converti en une machine à états finis $M(G')$, il n'est pas utile de tenir compte de l'utilisabilité de G, qui peut donc être contextuelle.
2. Nous ne voulons pas que $L(G')$ accepte des séquences de terminaux interdites par $L(G)$. L'algorithme d'approximation doit donc générer G' telle que

   $$L(G') \subset L(G)$$

3. Nous désirons une approximation régulière compréhensible, que l'auteur de la grammaire G puisse contrôler. L'approximation régulière est donc une *grammaire* sauvegardée dans un fichier qui, dans un second temps, sera compilée en une machine à états finis.
4. Etant donné P, une phrase appartenant à $L(G')$, $M(G')$ doit non seulement accepter P, mais également y insérer les informations nécessaires à la construction de son arbre syntaxique.

Phases. Afin de tenir compte de ces principes généraux, nous proposons une approche comportant 3 phases distinctes :

1. Approximation régulière de G en G'.
2. Génération d'un fichier Ovide à partir de G', et compilation de ce fichier. Le résultat est $M(G')$.
3. Pour une phrase P donnée,
   (a) Composition de $A(P)$ avec $M(G')$.
   (b) Construction de l'arbre syntaxique $Tree(P)$ à partir du résultat de la composition précédente.

Format. Une phrase est une séquence linéaire de terminaux de la grammaire. Son analyse, sous la forme d'une machine à états finis, sera donc également linéaire. Or, l'analyse syntaxique à générer est un arbre syntaxique. Afin de pouvoir générer cet arbre à partir de cette machine linéaire, les différents niveaux de l'arbre doivent donc y être représentés par *parenthésage*.

La Figure 18.2 illustre le principe, sur la séquence « $Det\ N\ V\ Det\ N$ », dont l'arbre syntaxique a été présenté en Figure 18.1. Cette machine est un transducteur, qui ne présente des non-terminaux que sur la couche d'entrée. Sur cette même couche figure en outre un

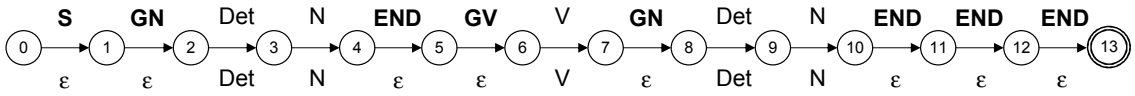


FIG. 18.2: Approximation régulière : analyse linéaire parenthésée. La séquence analysée est « *Det N V Det N* »

nouveau symbole, `END`, dont le rôle est d'indiquer la fin du dernier non-terminal rencontré, dans un parcours linéaire depuis l'état initial de la machine en direction de l'état final. C'est ce nouveau symbole qui permet de représenter, dans la machine linéaire, le parenthésage des différents niveaux de non-terminaux de la phrase, depuis le niveau le plus proche des terminaux, jusqu'à l'axiome de la grammaire. C'est ce que nous appelons une *analyse linéaire parenthésée*.

Définition 5.1 (Analyse linéaire parenthésée)**.** *Etant donné P, une séquence valide de terminaux appartenant à une grammaire G donnée, l'*analyse linéaire parenthésée *de P est elle-même une séquence P' où toute sous-séquence de P, correspondant à un syntagme valide de G, est entourée à gauche par le non-terminal du syntagme en question, et à droite par le non-terminal `END`.*

Le fait que les non-terminaux soient présents sur la couche d'entrée de l'analyse est une conséquence des étapes menant à la construction de $M(G')$, que nous présentons ci-après. Cette insertion des non-terminaux, en entrée du résultat, implique que la phrase soit le deuxième argument de la composition avec la grammaire :

$$T(P) = M(G') \circ A(P)$$

où $T(P)$ est le transducteur qui représente l'analyse linéaire parenthésée, à partir duquel $Tree(P)$ sera construit.

Grammaires gérées. Dans nos principes généraux, nous avons mentionné l'intention de gérer des grammaires contextuelles. Actuellement, l'algorithme accepte uniquement les grammaires contextuelles de la forme

$$l\,X\,r \rightarrow l\,w\,r \quad \text{avec} \quad \begin{array}{l} X \in N \\ w \in (N \cup T)^+ \\ l, r \in T^* \end{array}$$

En d'autres termes, nous n'acceptons pas, actuellement, de non-terminaux dans les contextes des règles.

Syntaxe de la grammaire. Les règles acceptées par l'algorithme d'approximation régulière doivent respecter la syntaxe suivante :

$$l\,[X]\,r \rightarrow w$$

A partir de cette syntaxe, l'algorithme construit lui-même des règles de forme classique :

$$l\,X\,r \rightarrow l\,w\,r$$

En outre, lorsque plusieurs règles présentent la même partie gauche, comme

$$l\,[X]\,r \rightarrow w$$
$$l\,[X]\,r \rightarrow y$$
$$l\,[X]\,r \rightarrow z$$

l'algorithme veille à ne créer qu'une seule règle, où les différentes réécritures sont réunies par union :

$$l\,X\,r \rightarrow l\,(\,w \mid y \mid z\,)\,r$$

Approximation régulière. L'algorithme est présenté en Pseudocode 38 et est illustré par les règles suivantes :

$$\begin{array}{lllll} \text{1a.} & a\,[S]\,c & \rightarrow & B \\ \text{1b.} & a\,[S]\,c & \rightarrow & b\,S\,B \\ \text{1c.} & a\,[S]\,c & \rightarrow & b \\ \text{2.} & a\,[B]\,b\,c & \rightarrow & X\,C\,y \end{array}$$

où les symboles majuscules identifient des non-terminaux, et les minuscules, des terminaux.

L'approximation réalisée dépend d'une profondeur d précisée en argument. On constate que le nouvel axiome de la grammaire n'est plus S, mais $S0$ (ligne 1). L'alphabet des non-terminaux, N', est quant à lui initialisé comme l'union de N et du symbole `END` (ligne 2), et est complété par les versions indexées des symboles de N, les indices allant de 0 à $d-1$ (lignes 3-5) : $S0, S1, \ldots, Sd-2, Sd-1$. L'algorithme utilise une structure H (ligne 6), qui mémorise les éléments nécessaires à la construction des règles non-récursives. Le reste de l'algorithme est consacré à la construction de ces règles (lignes 7-32).

Pour une règle donnée, la première étape est de détecter les différentes parties qui la constituent (ligne 9). Par exemple, pour la règle 2 :

$$\begin{array}{ll} B & \text{: cible} \\ X\,C\,y & \text{: réécriture} \\ a & \text{: contexte gauche} \\ b\,c & \text{: contexte droit} \end{array}$$

Pour chaque profondeur acceptée, l'approximation crée une nouvelle règle constituée de non-terminaux non-récursifs. De manière générale, l'idée est que, si la cible est un non-terminal de niveau i, la réécriture contiendra des non-terminaux de niveau $i+1$. La méthode qui assure cette réécriture est `Replace`, qui est toujours appliquée à la cible (ligne 11).

Require: $G = (S, N, T, R)$, une grammaire hors-contexte ; d, une profondeur
Ensure: La grammaire régulière $G' = (S', N', T, R')$, de profondeur d

```
1: S' ← S0
2: N' ← N ∪ {END}
3: for i ← 0 to (d−1) do
4:   for each nt ∈ N do N' ← N' ∪ {nt · i} end for
5: end for
6: H ← ∅
7: R' ← ∅
8: for each rule ∈ R do
9:   (target, rewrite, left, right) ← SplitRule(rule)
10:  for i ← 0 to (d−1) do
11:    targetI ← Replace(target, i, N)
12:    if target = "S" and BELONG(rewrite, "S") = FALSE then
13:      rewriteI ← Replace(rewrite, i, N)
14:    else
15:      if i < (d−1) then
16:        rewriteI ← Replace(rewrite, (i+1), N)
17:      else if Contain(rewrite, N) then
18:        rewriteI ← ϵ
19:      end if
20:    end if
21:    upper ← MakeUpper(left, targetI, right)
22:    if (p ← GET(H, upper)) = NULL then
23:      p ←(target, rewrite, left, right, w)
24:      INSERT(H, upper, p)
25:    else
26:      p.rewrite ← "p.rewrite | rewrite"
27:    end if
28:  end for
29: end for
30: for each (upper, p) ∈ H do
31:  r' ← "upper → p.left p.target p.rewrite END p.right / p.w"
32:  R' ← R' ∪ r'
33: end for
34: G' ←(S', N', T, R')
35: return G'
```

Pseudocode 38: `RegularApproximation`

Cette méthode remplace, dans la variable qu'elle reçoit, tout non-terminal par le même non-terminal, indexé selon l'indice fourni. Par exemple,

$\texttt{Replace}(X\,C\,y, 2, N)$

aura pour résultat :

$X2\,C2\,y$

La modification de la réécriture est plus délicate (lignes 12-20). Un premier test (lignes 12-13) permet de traiter les règles dont la cible est l'axiome, et dont la réécriture *ne contient pas* l'axiome. Dans ce cas, la réécriture est modifiée par `Replace` avec le même niveau de profondeur que la cible. Ceci évite de perdre inutilement un niveau, lorsque la réécriture de l'axiome n'implique pas l'axiome lui-même. Par exemple, pour la profondeur 1, la règle

$a\,[S]\,c \rightarrow B$

sera réécrite

$a\,S1\,c \rightarrow a\,B1\,c$

Dans tous les autres cas (lignes 15-19), le traitement réalisé dépend de la profondeur en cours. Pour une profondeur inférieure à $d-1$, `Replace` descend d'un niveau dans la profondeur (ligne 16). Par exemple, pour la profondeur 1, la règle

$a\,[S]\,c \rightarrow b\,S\,B$

sera réécrite

$a\,S1\,c \rightarrow a\,b\,S2\,B2\,c$

Si la profondeur traitée est $d-1$ et que la réécriture contient encore des non-terminaux, la réécriture est simplement remplacée par ϵ (lignes 18), ce qui met un terme à la récursivité. Par exemple, pour $d = 4$, si la profondeur en cours vaut 3, la règle

$a\,[S]\,c \rightarrow b\,S\,B$

sera réécrite

$a\,S3\,c \rightarrow a\,\epsilon\,c$

L'étape suivante est de créer la partie haute de la règle, constituée du contexte gauche, de la cible et du contexte droit (ligne 21). Les différentes parties nécessaires à la construction d'une règle complète sont ensuite sauvegardées dans la structure H, classée sur la partie haute (lignes 22-27) : la cible, la réécriture, le contexte gauche, le contexte droit et un poids. Si la partie haute existe déjà, seule la réécriture associée est mise à jour : il y a union de la réécriture existante et de la nouvelle réécriture.

Quatre types de poids peuvent être utilisés :

1. Soit des poids décroissants, ce qui favorise les structures profondes.
2. Soit des poids croissants, ce qui favorise les structures larges.
3. Soit un poids de 1, quelle que soit la profondeur, ce qui favorise simplement les structures contenant peu de syntagmes, tous niveaux confondus.
4. Soit, enfin, des poids estimés par apprentissage sur un corpus d'entraînement, ce qui favorise les structures les plus fréquentes. Nous revenons sur ceci, en Section 5.1.4, dans un paragraphe qui y est consacré.

Voici un exemple d'élément mémorisé dans la structure H. Cet élément correspond aux valeurs des règles 1a, 1b et 1c, pour une profondeur 0 et un poids de 1. Dans cet élément, la réécriture a été complétée progressivement :

partie haute : $a\,S0\,c$

• *cible :*	S
• *réécriture :*	$B0 \mid b\,S1\,B1 \mid b$
• *contexte gauche :*	a
• *contexte droit :*	c
• *poids :*	1

L'algorithme se termine par la création de l'ensemble de règles (lignes 31-32). Lorsqu'une règle est créée, la partie gauche est construite de sorte que la réécriture soit entourée à gauche par la cible, et à droite par le symbole `END`, afin de permettre l'identification et la délimitation du non-terminal qui a été réécrit. L'application de ce principe à partir de la structure présentée ci-dessus donne le résultat suivant :

$$a\,S0\,c \rightarrow a\,\mathbf{S}\,(B0 \mid b\,S1\,B1 \mid b)\,\mathbf{END}\,c\ /\ 1$$

Il est maintenant possible d'identifier la cible qui a été réécrite, et de distinguer ses limites dans la partie droite de la règle, étant donné que les contextes sont à l'extérieur des bornes insérées.

L'application de l'ensemble de l'algorithme sur les règles de notre exemple, pour une profondeur $d = 4$ et avec un poids décroissant, donne le résultat suivant :

$$\begin{array}{lcl}
a\,S0\,c & \rightarrow & a\,\mathbf{S}\,(\,B0\,b \mid b\,S1\,B1\,b \mid b\,)\,\mathbf{END}\,c\ /\ 4 \\
a\,S1\,c & \rightarrow & a\,\mathbf{S}\,(\,B1\,b \mid b\,S2\,B2\,b \mid b\,)\,\mathbf{END}\,c\ /\ 3 \\
a\,S2\,c & \rightarrow & a\,\mathbf{S}\,(\,B2\,b \mid b\,S3\,B3\,b \mid b\,)\,\mathbf{END}\,c\ /\ 2 \\
a\,S3\,c & \rightarrow & a\,\mathbf{S}\,(\,B3\,b \mid b\,)\,\mathbf{END}\,c\ /\ 1
\end{array}$$

$$\begin{array}{lcl}
a\,B0\,b\,c & \rightarrow & a\,\mathbf{B}\,X1\,B1\,y\,\mathbf{END}\,b\,c\ /\ 4 \\
a\,B1\,b\,c & \rightarrow & a\,\mathbf{B}\,X2\,B2\,\mathbf{END}\,y\,b\,c\ /\ 3 \\
a\,B2\,b\,c & \rightarrow & a\,\mathbf{B}\,X3\,B3\,\mathbf{END}\,y\,b\,c\ /\ 2 \\
a\,B3\,b\,c & \rightarrow & a\,\mathbf{B}\,\epsilon\,\mathbf{END}\,b\,c\ /\ 1
\end{array}$$

L'algorithme d'approximation convertit donc une grammaire contextuelle en une grammaire régulière, qui diffère de la grammaire initiale en ceci qu'elle limite la récursivité à une profondeur d. Cette grammaire, en outre, présente des bornes permettant d'identifier les non-terminaux réécrits et leurs limites dans les différentes réécritures. Il reste dès lors à convertir cette grammaire régulière en une machine à états finis.

De la grammaire régulière à la machine à états finis. Lors de l'analyse d'une phrase, le résultat que nous désirons obtenir est un transducteur dont le format soit celui présenté en Figure 18.2. Il est donc nécessaire de compiler la grammaire régulière en un transducteur qui assure ce format. Pour ce faire, nous utilisons notre compilateur Ovide.

Nous illustrons la mise en œuvre de la compilation sur la grammaire hors-contexte présentée en Table 18.1.

Afin d'obtenir le résultat désiré, 3 fichiers Ovide sont nécessaires. Ils utilisent le même alphabet, qui correspond à $(N' \cup T)$. Le fichier-phare de la compilation est celui contenant les règles de G', que nous appelons `grammar-rule`. Nous commençons par décrire le format de ce fichier, de manière à mettre en évidence les raisons qui justifient le recours aux deux autres fichiers.

Fichier `grammar-rule`. L'objectif de ce fichier est de décrire le langage accepté. C'est donc lui qui contient, dans une section [`RULE`], les règles de la grammaire régulière précédemment construite. Ce fichier tient en outre compte des contraintes suivantes :

1. Par défaut, Ovide accepte le langage Σ^*, qui correspond, dans ce cas-ci, à $(N' \cup T)^*$. Sans autre précision, Ovide appliquerait donc les règles de la grammaire au langage par défaut. Or, l'objectif de la grammaire est de n'accepter que les séquences de terminaux correspondant à des dérivations valides depuis *l'axiome de la grammaire*. Le langage accepté doit donc, *a priori*, être réduit à $S0$, l'axiome de G'. Cependant, il est fréquent que les phrases analysées soient a-grammaticales, du fait de ruptures de construction. Dans cette optique, le langage que nous acceptons est élargi à $(S0)^*$, ce qui permet d'analyser une phrase a-grammaticale sous la forme de plusieurs petites phrases valides.
2. Les règles de réécriture doivent toujours être ordonnées, afin d'assurer l'exactitude de la réécriture réalisée. En ce qui concerne les règles de la grammaire, deux contraintes doivent être respectées :
   (a) Les règles d'un niveau i doivent toujours être situées avant les règles du niveau i+1. Ceci permet d'assurer que les non-terminaux d'un niveau donné soient effectivement réécrits par les règles du niveau suivant.
   (b) Dans un niveau de règles donné, les règles concernant l'axiome doivent toujours être situées avant les autres règles du même niveau, parce que la réécriture de l'axiome utilise des non-terminaux du même niveau.

Le fichier `grammar-rule`, qui respecte les contraintes ci-dessus, est illustré en Figure 18.3. La compilation de ce fichier donne un transducteur qui accepte en entrée le langage

```
S    →  GS GV | GN | GV
GS   →  GN | Pron
GNP  →  Prep GN
GND  →  De GN
GN   →  Det Adj{,2} N Adj? | Det Adj | N | Propername | GN GND
GN   →  GN Pronrel S | Pron Pronrel S
GV   →  V GNP | V GN GNP | V GNP GN
GV   →  PronCI? V GN | PronCD V GNP | PronCD? PronCI? V
```

TAB. 18.1: Exemple de grammaire hors-contexte très simple

$(S0)^*$, et en sortie l'ensemble des séquences valides telles que définies par G'. Chaque séquence valide est une analyse linéaire parenthésée, selon la Définition 5.1 : elle est donc constituée de symboles appartenant à $(T \cup N \cup \texttt{END})$. Tel quel, ce transducteur est inutilisable, et ce pour deux raisons :

1. La couche d'entrée de la machine, qui ne contient que le langage $(S0)^*$, ne nous intéresse pas. Elle est exclusivement nécessaire au moment de la compilation, afin de limiter le langage accepté. Après compilation, seule la sortie du transducteur nous intéresse : c'est elle qui présente l'analyse linéaire parenthésée. Le transducteur peut donc être réduit au transducteur identitaire correspondant à sa seconde projection.
2. Le transducteur, réduit à sa seconde projection, ne peut cependant être composé avec une phrase à analyser : la phrase ne contient que des terminaux, alors que le transducteur présente des non-terminaux. Il est donc nécessaire de faire disparaître les non-terminaux de la sortie du transducteur, afin de permettre la composition de la grammaire avec une phrase.

Les autres fichiers. Les deux manipulations présentées ci-dessus doivent donc être réalisées sur le transducteur correspondant à la compilation de `grammar-rule`. Nous attirons l'attention sur le fait que la seconde projection *doit* être réalisée *avant* la suppression des non-terminaux de la sortie du transducteur. Dans le cas contraire, l'analyse linéaire parenthésée serait perdue. Ceci explique le recours à deux autres fichiers Ovide :

1. Le fichier `grammar-delN`, qui supprime les non-terminaux sur la sortie du transducteur. Ce fichier est illustré en Figure 18.4.
2. Un fichier général, qui inclut `grammar-rule` et `grammar-delN`, et définit le mode de compilation à leur appliquer. Nous l'appelons `grammar`, parce que c'est le résultat de sa compilation qui correspond à la machine $M(G')$ désirée. Ce fichier est illustré en Figure 18.5. La règle de compilation définie dans la section [`COMPILE`] donne le résultat voulu, du fait de la précédence de l'opérateur de seconde projection (`>o`) sur l'opérateur

de composition ($^\circ$). Pour toute information complémentaire concernant le compilateur, nous renvoyons le lecteur à la documentation (cf.Annexe B).

Du transducteur à l'arbre syntaxique. Nous rappelons que la composition d'une phrase P avec la grammaire G' est réalisée comme suit :

$$T(P) = M(G') \circ A(P)$$

Le résultat, $T(P)$, contient l'analyse linéaire parenthésée de P. Sur la couche d'entrée de $T(P)$, tout syntagme est précédé de son-terminal et est suivi du symbole `END`, et sur la couche de sortie, tout non-terminal vaut ϵ. Ces trois informations facilitent la définition d'un algorithme construisant l'arbre syntaxique $Tree(P)$ à partir de $T(P)$.

L'algorithme `CreateTree` est présenté en Pseudocode 39. Il repose sur deux structures de données. D'une part, un vecteur V, qui mémorise, pour tout état de $T(P)$, le nœud de $Tree(P)$ qui y correspond (lignes 4, 14-17 et 24-27). D'autre part, une pile C, dont le sommet référence toujours l'*état* auquel commence le syntagme en cours de construction. Cet état correspond, *via* V, à un nœud de l'arbre. C'est à ce nœud que sera rattaché tout élément qui en dépend *directement*. Tout ajout dans l'arbre (lignes 18-19 et 29-30) se fait donc systématiquement *après* avoir déterminé l'état qui se trouve en sommet de la pile (lignes 12 et 28).

Ceci étant posé, le reste de l'algorithme est fort simple. La pile est initialisée avec l'état initial du transducteur (ligne 3), qui correspondra à la racine de l'arbre (lignes 4-5). A chaque itération, on analyse les symboles de *la seule transition* qui quitte l'état courant (ligne 9). L'itération est assurée par l'évolution de la valeur de l'état courant, mise à jour par la valeur de l'état atteint pas la transition (ligne 32).

Le premier test concerne le symbole de sortie (ligne 10). S'il s'agit de ϵ, le symbole d'entrée est un non-terminal (lignes 11-22). Si ce symbole n'est pas le symbole `END`, cela signifie que l'on descend d'un niveau dans la profondeur de l'arbre (lignes 12-19). De ce fait, on empile l'état atteint par la transition (ligne 13). Le non-terminal, nécessaire à l'analyse, devient une nouvelle branche de l'arbre (lignes 18-19). Si le symbole d'entrée est `END`, cela signifie qu'on remonte d'un niveau dans la profondeur de l'arbre. De ce fait, on dépile (ligne 21). Ce symbole, qui ne présente aucun intérêt pour l'analyse, n'est pas ajouté.

Lorsque le symbole courant est un terminal, aucune action n'est réalisée sur la pile, hormis la consultation de son sommet (ligne 28). Le symbole terminal devient une nouvelle branche de l'arbre (lignes 29-30).

La Figure 18.6 donne un exemple de transducteur correspondant à une analyse linéaire parenthésée, et la Figure 18.7 illustre le résultat obtenu par application de `CreateTree` sur ce transducteur. La phrase analysée est « La petite de la ferme blanche rend le cartable de Luc à son père ». La grammaire utilisée est celle de la Table 18.1.

Require: $T = (\Sigma, \Sigma, Q, i, F, E)$, un transducteur
Ensure: $Tree = (\Sigma, Q', i', E')$, l'arbre (syntagmatique) étiqueté correspondant à T

```
1: V[p] ← NULL for p ← 0 to |Q|
2: E' ← C ← ∅
3: ENQUEUE(C, i)
4: V[i] ← i' ← 0
5: Q' ←{i'}
6: p ← i
7: n ← 1
8: while p ∈ (Q−F) do
9:    e ← E[p][0]
10:   if e.a2 = ϵ then
11:      if e.a1 ≠ END then
12:         p ← TOP(C)
13:         ENQUEUE(C, e.q)
14:         if V[e.q] = NULL then
15:            V[e.q] ← n
16:            n ← n+1
17:         end if
18:         Q' ← Q' ∪ {V[e.q]}
19:         E' ← E' ∪ {(V[p], e.a1, V[e.q])}
20:      else
21:         DEQUEUE(C)
22:      end if
23:   else
24:      if V[e.q] = NULL then
25:         V[e.q] ← n
26:         n ← n+1
27:      end if
28:      p ← TOP(C)
29:      Q' ← Q' ∪ {V[e.q]}
30:      E' ← E' ∪ {(V[p], e.a1, V[e.q])}
31:   end if
32:   p ← e.q
33: end while
34: Tree ←(Σ, Q', i', E')
35: return Tree
```

Pseudocode 39: `CreateTree`

```
[CLASSIN]
  EPS \x00
[LANGIN]
  (S0)*
[RULE]
  S0 → S (GS0 GV0 | GN0 | GV0) END
  GN0 → GN (Det Adj{,2} N Adj? | Det Adj | ... | GN1 GND1 | ...) END
  GND0 → GND De GN1 END
  ...
  S1 → S (GS1 GV1 | GN1 | GV1) END
  GN1 → GN (Det Adj{,2} N Adj? | Det Adj | ... | GN2 GND2 | ...) END
  GND1 → GND De GN2 END
  ...
  S8 → S (GS8 GV8 | GN8 | GV8) END
  GN8 → GN (Det Adj{,2} N Adj? | Det Adj | N | Propername) END
  GND8 → GND <EPS> END
```

FIG. 18.3: Fichier `grammar-rule` : langage et règles correspondant à G'

```
[CLASSIN]
  EPS \x00
[RULE]
  [S GS GN GV GNP GND] → <EPS>
  END → <EPS>
```

FIG. 18.4: Fichier `grammar-delN` : suppression des non-terminaux appartenant à ($N \cup$ `END`)

```
[INCLUDE]
  grammar-rule
  grammar-delN
[COMPILE]
  (@grammar-rule)>o ° @grammar-delN
```

FIG. 18.5: Fichier `grammar` : définition d'une compilation particulière

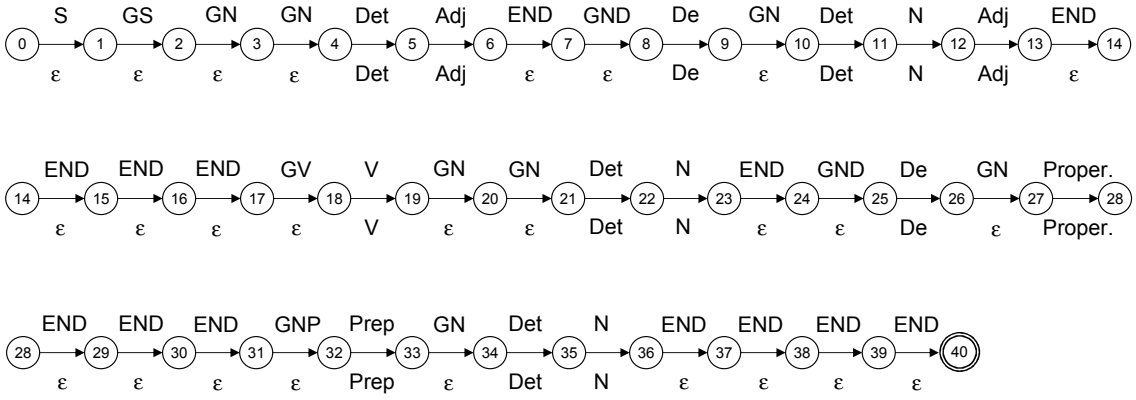


FIG. 18.6: Analyse linéaire parenthésée, à partir de la grammaire présentée en Table 18.1, pour la phrase :« La petite de la ferme blanche rend le cartable de Luc à son père ». La machine a été découpée, pour une plus grande lisibilité

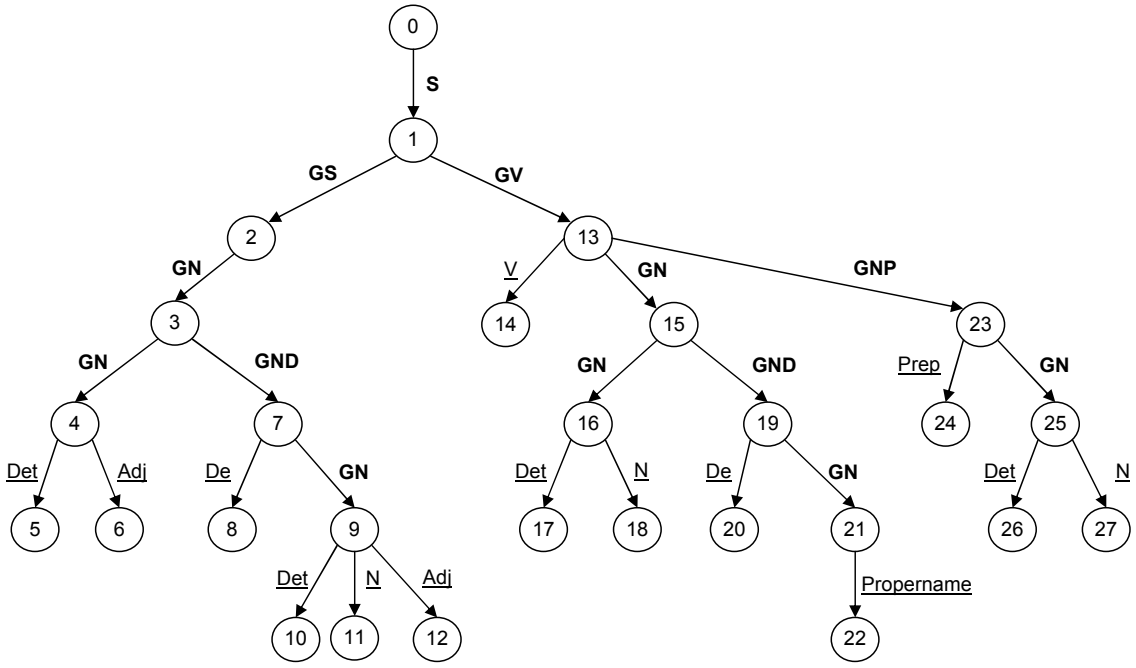


FIG. 18.7: Arbre syntaxique correspondant à l'analyse linéaire parenthésée de la Figure 18.6

5.1.4 Questions ouvertes

Nous mentionnions, en début de section précédente, que l'algorithme d'approximation régulière n'accepte pas, actuellement, les grammaires contextuelles dont les contextes contiennent des non-terminaux. L'algorithme, en l'état, suscite d'autres questions.

Comment gérer la précédence des règles ? Lorsque plusieurs règles concernent la même cible et que leurs contextes ne sont pas mutuellement exclusifs, comment les ordonner dans le fichier Ovide ? Imaginons par exemple les règles

$$a\,[S] \rightarrow A$$
$$[S]\,b \rightarrow B$$

Si la phrase en cours d'analyse présente la séquence $a\,S\,b$, quelle règle doit être appliquée ? Uniquement la première règle rencontrée ? Les deux règles ?

Une grammaire de quelle profondeur ? Est-il bon de compiler un seul transducteur représentant l'ensemble des profondeurs acceptées ? Nous avons constaté, sur l'ensemble de règles de la Table 18.1, que l'augmentation de la profondeur a une influence considérable sur la taille du transducteur compilé (cf. Table 18.2). En arrondissant, l'ajout d'un niveau multiplie par 3 la taille du transducteur. Pourtant, la grammaire considérée est simpliste... Dès lors, ne vaut-il pas mieux compiler les différentes profondeurs séparément, pour ne les composer ensemble qu'au moment de l'analyse d'une phrase ? Cette stratégie permettrait certainement de gagner de la place... mais ne serait-ce au détriment du temps de calcul ?

En outre, l'algorithme repose sur une profondeur d pré-déterminée. Comment déterminer cette profondeur, afin d'assurer que l'ensemble des productions linguistiques valides puissent être acceptées par le transducteur ? Un apprentissage pourrait-il répondre à cette question ?

Profondeur	**Taille (octets)**
1	826
2	3 218
3	10 594
4	33 362
5	105 098
6	329 538
7	1 033 314
8	3 238 794

Tab. 18.2: Evolution de la taille des grammaires compilées

Peut-on apprendre une grammaire ? En se limitant au modèle de langue, il est évident qu'une grammaire peut être apprise : il est assez facile d'obtenir un corpus de quelques centaines de milliers de mots accompagnés de leurs catégories grammaticales. Cependant, malgré la taille des corpus disponibles, les données manquent. Il est de ce fait nécessaire de lisser le modèle de langue, de manière à ce qu'il puisse gérer les nombreux cas, rencontrés couramment lorsque le système est exploité, alors qu'ils ne l'ont pas été au cours de l'entraînement...

Etant donné que les mêmes séquences de catégories peuvent correspondre à différentes structures syntaxiques, le corpus nécessaire à l'estimation de l'ensemble des structures syntaxiques d'une langue est certainement bien plus important que celui permettant d'estimer les n-grammes grammaticaux possibles. Il nous semble dès lors difficile de construire une grammaire par apprentissage, parce que la grammaire apprise sera par nature *incomplète*.

Nous posions ci-dessus la question de savoir si la profondeur nécessaire, pour que la grammaire accepte l'ensemble des phrases valides, pouvait être estimée à l'aide d'un apprentissage. L'analyse précédente met en évidence qu'un tel apprentissage n'identifierait, *au mieux*, que la profondeur maximale d'une grammaire incomplète.

Sur la base de ces quelques réflexions, il nous semble que la construction d'une grammaire, mais aussi la décision de la profondeur maximale qu'une grammaire régulière devrait accepter, restent du ressort de l'expertise linguistique. L'utilisation d'une grammaire, capable de vérifier la validité de la structure syntaxique d'une phrase, augmentera donc certainement les ressources humaines nécessaires à la mise en place d'un système multilingue, puisqu'on peut supposer que chaque langue ajoutée au système nécessitera les compétences d'un expert différent.

Une grammaire experte peut cependant certainement tirer avantage d'une pondération des règles qui la composent, à partir de données d'entraînement. Comme le note Eisner (1996a), il est raisonnable d'attendre d'une structure syntaxique donnée qu'elle soit aussi présente dans les données de test que dans les données d'entraînement. C'est sur la base de ce postulat qu'ont été proposées des méthodes permettant d'estimer des grammaires hors-contexte probabilistes, les PCFG [2] (Charniak 1993, Johnson 1998b). Nos futurs travaux s'inscriront dans cette lignée.

Une grammaire « aveugle » est-elle suffisante ? L'ébauche d'analyse syntaxique présentée dans cette section ignore les mots de la phrase, pour ne considérer que les catégories de ces mots. Or, les recherches dans le domaine des PCFG ont mis en évidence le besoin de l'intégration des formes lexicales dans l'estimation de la structure syntaxique. Les chercheurs ont ainsi étendu leurs grammaires d'informations lexicales contextuelles, sous la forme soit de bi-grammes lexicaux, soit de dépendances lexicales fixant les structures valides correspondant à chaque mot du lexique (Black *et al.* 1993, Collins 1996, Eisner 1996a,b, 2002b, Eisner & Satta 1999). Cette piste est certainement à suivre.

[2] PCFG : *Probabilistic Context-Free Grammar*

Notons qu'une première étape, dans notre cas, sera certainement de combiner l'analyse syntaxique parenthésée au modèle de langue que nous avons présenté en analyse syntaxique. Ce modèle a au moins l'avantage d'estimer, au travers du modèle d'ambiguïté lexicale, la probabilité d'association d'une paire (*mot*,*catégorie*).

Ces quelques réflexions recevront toutes notre attention au cours de nos travaux futurs, dans le cadre de l'établissement d'une grammaire adaptée à la correction flexionnelle.

5.2 Gestion des mots hors-vocabulaire

Dans ce domaine particulier de l'analyse morphologique, l'objectif de cette thèse a été de proposer des modèles de correction capables de tenir compte des caractéristiques de la forme. C'est sur cette base que le modèle typographique et le modèle phonétique ont été définis.

La gestion des mots hors-vocabulaire ne peut cependant se contenter de modèles de correction : de nombreuses formes sont correctes, mais inconnues de nos lexiques. Pour remédier à ce problème, nos modèles de correction se sont vus complétés par un modèle de casse, qui permet de détecter les noms propres et les acronymes, et un modèle morphologique, qui détermine les catégories possibles des mots hors-vocabulaire en fonction de leur terminaison.

L'algorithme proposé limite cependant fortement l'influence du modèle morphologique : celui-ci n'intervient que si les autres modèles n'ont donné *aucun résultat*. De nombreuses formes correctes, mais inconnues de nos lexiques, sont de ce fait corrigées, sans que le modèle morphologique n'ait eu l'occasion de les analyser.

Pour éviter cet écueil, le modèle morphologique devrait être le premier à analyser une forme inconnue. Remonter le modèle morphologique dans l'algorithme d'analyse n'est pourtant pas envisageable *en l'état*. En effet, le fait que ce modèle ne valide pas *l'ensemble de la forme*, mais ne s'intéresse qu'à sa *terminaison*, ne permet pas de lui accorder une telle importance dans le processus d'analyse. Déplacer le modèle morphologique implique donc de le revoir en profondeur.

Nous proposons ci-dessous une typologie des formes inconnues que l'approche actuelle ne gère pas, et les principes d'analyse que nous pensons leur appliquer prochainement. Sur cette base, nous présentons ensuite un algorithme d'analyse possible, tenant compte de l'ensemble des formes hors-vocabulaire.

5.2.1 Typologie des formes inconnues

Les mots absents et les néologismes. Les mots absents sont des mots appartenant au lexique de la langue, mais absents du dictionnaire utilisé. Il est évident que certains dictionnaires sont plus complets que d'autres, mais de manière générale, aucun dictionnaire ne possède la totalité des lexèmes de la langue. Notre dictionnaire, par exemple, contient

très peu de lexèmes en *-isation* (*latinisation*, *romanisation*, *sacralisation*, etc.).

Contrairement aux mots absents, les néologismes sont des formes nouvelles [3], obtenues principalement par dérivation [4] et par contraction [5]. Rey (1995) définit le néologisme comme un mot dont la relation signifiant/signifié n'a, auparavant, jamais été matérialisée dans le lexique de la langue.

Mots absents et néologismes partagent un point commun : ils respectent les règles dérivationnelles et flexionnelles de la langue. De ce fait, nous proposons de leur appliquer une analyse morpho-syllabique, dont l'objectif sera de valider la forme dans son ensemble. La première étape de cette analyse serait certainement le modèle morphologique actuel, chargé de détecter la plus longue terminaison possible. Nous envisageons ensuite trois approches possibles :

1. Une analyse morphologique, qui devrait repérer les morphèmes potentiellement présents dans la forme, et déterminer si les règles flexionnelles et dérivationnelles de la langue ont été respectées.
2. Plus simplement, une analyse syllabique pourrait déterminer, à l'aide de règles de bonne formation syllabique, si la forme est constituée d'une suite valide de syllabes de la langue.
3. Une approche mixte consisterait à valider la structure syllabique de la forme et le positionnement de certains morphèmes (préfixes, suffixes) identifiés dans la forme.

Ces trois approches sont par nature représentables par machines à états finis puisque, comme l'ont démontré Chomsky & Schützenberger (1963), la morphologie d'une langue n'implique pas d'imbrications infinies.

Les emprunts. Un emprunt est un terme intégré au lexique de la langue, alors qu'il appartient à une langue étrangère. Un emprunt peut être lexicalisé, c'est-à-dire adapté au système phonologique (*riding coat* → *redingote*) et/ou morphologique (*des pizza*s, *je dumpe*). Les emprunts lexicalisés au niveau phonologique appartiennent de longue date au lexique de la langue. Nous nous intéressons de ce fait ici uniquement aux emprunts non lexicalisés, ou dont la lexicalisation ne concerne que le niveau morphologique.

Ces formes demandent probablement une analyse en deux temps. Dans un premier temps, notre modèle d'analyse morphologique pourrait être appliqué. Ce modèle permettrait de déterminer si la forme a été morphologiquement lexicalisée, et d'évaluer les catégories possibles à attribuer à l'emprunt. Dans un second temps, la partie non lexicalisée devrait subir un test d'appartenance aux langues connues du système. Ce test peut être facilement réalisé à l'aide d'un modèle de langue entraîné sur des n-grammes de lettres appartenant

[3] Les néologismes sémantiques sont hors de notre propos, étant donné que le sens, ici, n'est pas pris en compte. La gestion des formes hors-vocabulaire se base sur des critères exclusivement morphologiques.

[4] Par exemple, *alunir*, dérivé de *lune* sur le modèle d'*atterrir*.

[5] Par exemple, *adulescent*, contraction de *adulte* et de *adolescent*, qui désigne un adulte qui conserve un comportement d'adolescent.

aux langues gérées par le système. Ce modèle de langue peut être représenté à l'aide d'une machine à états finis pondérée, comme nous l'avons démontré en analyse syntaxique (cf. Section 15.5).

Les abréviations et les troncations. L'abréviation est le raccourcissement d'un mot, avec suppression éventuelle de voyelles (*Melle* pour *Mademoiselle*, *etc.* pour *et cætera*), tandis que la troncation est le procédé d'abrègement qui consiste à supprimer une ou plusieurs syllabes en début ou en fin de mot (*blème* pour *problème*, *intox* pour *intoxication*), parfois avec addition de la voyelle « *-o* » (*apéro* pour *apéritif*).

Le procédé de troncation est par nature régulier. Il peut donc être décrit à l'aide de règles de réécriture. Une abréviation, quant à elle, ne présente que des lettres de la forme abrégée. Dans le principe, une abréviation peut donc être complétée par des règles de réécriture autorisant exclusivement l'insertion de lettres dans la forme. Lors de la réécriture, les risques de confusion seront cependant présents :

1. Certaines formes tronquées sont fort proches de formes appartenant au dictionnaire (*démo*, *démon*). Comment dès lors choisir entre correction (*démon*) et désabrègement (*démonstration*) ?
2. Plusieurs formes du lexique peuvent correspondre à la même abréviation (*ps* pour *post scriptum* ?, *plus* ?, ... ?). Comment dès lors choisir la bonne réécriture au sein de plusieurs réécritures possibles ?

Il s'agit là de cas limites, qui semblent indiquer la nécessité d'une information sémantique. Cette information ne sera cependant pas suffisante, parce que les formes abrégées sont plus souvent le fait de l'*habitude* que de contraintes sémantiques... Actuellement, nous laissons donc ces deux questions ouvertes.

5.2.2 Agencement en algorithme

Le Pseudocode 40 présente les grandes lignes de l'algorithme que nous comptons mettre en œuvre afin de gérer les formes hors-vocabulaire. Dans le pseudocode,

- `CHECKCASE` est notre modèle de casse. Si l'analyse réussit, les catégories *nom propre* et *acronyme* sont pondérées.
- `CHECKCORR` représente nos modèles de correction orthographique (distances typographique et phonétique). Si la correction réussit, les corrections proposées sont pondérées.
- `CHECKEND` fait référence au modèle morphologique actuel, qui identifie la plus longue terminaison valide d'une forme donnée. La terminaison peut être vide (ϵ). Dans tous les cas, les catégories qui y sont associées sont pondérées.
- `CHECKMORPHO` est l'étape de validation morpho-syllabique dédiée aux mots absents et aux néologismes. Le retour est simplement booléen.

- `CHECKBORROW` teste l'appartenance d'une forme inconnue à l'une des langues connues du système. En cas de succès, la langue identifiée est retournée.
- `CHECKABREV` détermine si une forme est une abréviation ou une troncation. Si la réécriture réussit, les réécritures proposées sont pondérées.
- Le préfixe `DLS_` identifie les méthodes qui permettent d'interagir avec la structure de données de notre système de synthèse, présentée en Section 13.2.

La forme est *a priori* considérée comme hors-vocabulaire (ligne 1). La première analyse réalisée est la détection de la terminaison (ligne 2), qui sera utile à la validation morpho-syllabique et à la détection de la langue, mais également dans le cas où toutes les analyses tentées auront échoué.

La détection de la terminaison permet d'extraire un radical (ligne 3) qui, si la terminaison est vide, correspond à la forme elle-même.

C'est sur ce radical que sont testées la validation morpho-syllabique (ligne 4) et l'appartenance à la langue (ligne 6). On constate que le test d'appartenance à la langue n'est réalisé que si la validation morpho-syllabique échoue. Le postulat est qu'un mot appartient plus fréquemment à la langue du texte qu'à une autre langue. Si la validation morpho-syllabique réussit, la forme est considérée comme appartenant au vocabulaire (ligne 5). Si le test d'appartenance à la langue réussit, la langue identifiée est attribuée à la forme (ligne 8) [6], mais la forme reste hors-vocabulaire. Dans les deux cas, la forme reçoit les catégories associées à la terminaison (lignes 11–13).

Si les deux validations précédentes échouent, la forme est soumise, *en parallèle*, à la correction (ligne 15) et à la détection des abréviations (ligne 16). L'objectif est ici d'attribuer l'ensemble des réécritures obtenues (lignes 19–24), sans donner l'avantage à l'une ou l'autre des approches. Ceci se base sur les risques de confusion que nous avons identifiés à la section précédente. Si au moins l'une de ces approches réussit, la forme est considérée comme appartenant au vocabulaire (ligne 18).

Si les tests de réécriture n'ont pas abouti, les catégories associées initialement à la terminaison sont attribuées à la forme (lignes 26–28), qui reste cependant hors-vocabulaire.

Quel que soit le résultat des analyses précédentes, le test de casse peut être réalisé (ligne 31). Celui-ci n'aura aucun impact sur l'origine de la forme, mais permettra de lui attribuer d'autres catégories, si la casse l'autorise (lignes 32–34).

5.2.3 Analyse

L'approche initialement proposée était certainement trop dépendante du lexique. Cette nouvelle approche ne nie pas l'intérêt du lexique, mais tâche de tenir compte des nombreux phénomènes qui font des lexiques une pâle représentation de la richesse de la langue. La langue est productive, au contraire de la *norme*, et les nouveaux modèles évoqués

[6] En synthèse de la parole, ceci permettra d'adapter les règles de phonétisation utilisées.

dans cette section devraient permettre de dépasser la norme, pour gérer un peu plus la langue. Ceci, bien sûr, reste à valider au travers de tests représentatifs.

Notons cependant qu'une partie considérable de la production écrite met en œuvre de nouveaux codes linguistiques, qui s'écartent volontairement de la norme, naviguant entre la phonétique, le codé et le crypté. C'est le cas du langage SMS [7], par exemple. Toute application qui voudrait toucher cette littérature particulière devrait probablement s'écarter de manière décisive des contraintes de la norme, pour s'intéresser un peu plus à la raison d'être de la langue, qui est indéniablement de faciliter la communication entre individus, en véhiculant un *sens* de manière orale ou de manière écrite. Mais ceci, incontestablement, est un autre sujet...

[7] *Short Message Service.*

Require: *word*, un mot OOV appartenant à la couche Word de la DLS
Ensure: *word* est analysé, et au besoin corrigé. Son origine est connue

```
1: DLS_SETORIGIN(word, OOV)
2: (end, tabCat) ← CHECKEND(word)
3: root ←(word − end)
4: if (isOK ← CHECKMORPHO(root)) = TRUE then
5:   DLS_SETORIGIN(word, IV)
6: else if (lang ← CHECKBORROW(root)) ≠ NULL then
7:   isOK ← TRUE
8:   DLS_SETLANGUAGE(word, lang)
9: end if
10: if isOK = TRUE then
11:   for each (cat, w) ∈ tabCat do
12:     DLS_ADDCAT(word, cat, w)
13:   end for
14: else
15:   tabCorr ← CHECKCORR(word)
16:   tabAbrv ← CHECKABREV(word)
17:   if tabCorr ≠ NULL  or tabAbrv ≠ NULL then
18:     DLS_SETORIGIN(word, IV)
19:     for each (spell, cat, w) ∈ tabCorr do
20:       DLS_ADDSPELLING(word, spell, cat, w)
21:     end for
22:     for each (spell, cat, w) ∈ tabAbrv do
23:       DLS_ADDSPELLING(word, spell, cat, w)
24:     end for
25:   else
26:     for each (cat, w) ∈ tabCat do
27:       DLS_ADDCAT(word, cat, w)
28:     end for
29:   end if
30: end if
31: tabCat ← CHECKCASE(word)
32: for each (cat, w) ∈ tabCat do
33:   DLS_ADDCAT(word, cat, w)
34: end for
```

Pseudocode 40: `ManageOOV`

Bibliographie

Aho, A. (1990). Algorithms for finding patterns in strings. In *Handbook of theoretical computer science* (pp. 256–300). B.V., Amsterdam : Elseviers Science Publishers.

Aho, A., Hopcroft, J., & Ullman, J. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers, Principles, Techniques and Tools*. Addison-Wesley.

Ait-Mokhtar, S. & Chanod, J. (1997). Incremental finite state parsing. In *ANLP'97*.

Allauzen, C., Mohri, M., & Riley, M. (2004). Statistical modeling for unit selection in speech synthesis. In *Proceedings of ACL'04* (pp. 55–62). Barcelona, Spain.

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., & Mohri, M. (2007). OpenFst : A General and Efficient Weighted Finite-State Transducer Library. In J. Holub & J. Zdárek (Eds.), *Implementation and Application of Automata, 12th International Conference (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science* (pp. 11–23). : Springer.

Andreewsky, A. & Fluhr, C. (1973). Experience de constitution d'un programme d'apprentissage pour le traitement automatique du langage. In *Proceedings of the 5th conference on Computational linguistics* (pp. 109–121). Morristown, NJ, USA : Association for Computational Linguistics.

Angell, R., Freund, G., & Willett, P. (1983). Automatic spelling correction using a trigram similarity measure. *Information Processing and Management*, 19(4), 305–316.

Atwell, E. & Elliott, S. (1987). Dealing with ill-formed English text. In R. Garaside, G. Leach, & G. Sampson (Eds.), *The Computational Analysis of English : a Corpus-Based Approach* chapter 10. New York : Longman Inc.

Béal, M. & Carton, O. (2000). Computing the prefix of an automaton. *Theoretical Informatics and Applications*, 34(6), 503–514.

Balestri, M., Pacchiotti, A., Quazza, S., Salza, P., & Sandri, S. (1999). Choose the best to modify the least : a new generation concatenative synthesis system. In *Proceedings of Eurospeech'99* (pp. 2291–2294). Budapest, Hungary.

Baroni, M. & Bernardini, S. (2004). BootCaT : Bootstrapping corpora and terms from the web. In *Proceedings of LREC'04* (pp. 1313–1316).

Bassino, F., David, J., & Nicaud, C. (1999). REGAL : a library to randomly and exhausively generate automata. In *Proceedings of CIAA* (pp. 2291–2294). Budapest, Hungary.

Baum, L. (1972). An inequality and associated maximization technique in statistical estimation for probabilistic functions of markov processes. *Inequalities*, 3, 1–8.

Beaufort, R. (2006). *Compiling rewrite rules into finite-state transducers.* Technical report, Text-To Speech Group, Speech Processing Department, Multitel ASBL, `http ://www.multitel.be/TTS/`.

Beaufort, R. & Colotte, V. (2006). Method and device for speech synthesis. European Patent EP1640968.

Beaufort, R., Dutoit, T., & Pagel, V. (2002). Analyse syntaxique du français. Pondération par trigrammes lissés et classes d'ambiguïtés lexicales. In *Proceedings of JEP* (pp. 133–136).

Beaufort, R. & Mancas-Thillou, C. (2007). A Weighted Finite-State Framework for Correcting Errors in Natural Scene OCR. In *Proceedings of ICDAR'07* (pp. 889–893).

Beaufort, R. & Ruelle, A. (2006). eLite : système de synthèse de la parole à orientation linguistique. In *Proceedings of JEP* (pp. 509–512).

Beekes, R. (1995). *Comparative Indo-European Linguistics. An Introduction.* Amsterdam : Benjamin.

Beesley, K. & Karttunen, L. (2003). *Finite-State Morphology.* Center for the Study of Language and Information. 503 pages.

Bellman, R. & Dreyfus, S. (1962). *Applied Dynamic Programming.* New Jersey : Princeton University Press.

Bentley, J. (1985). A spelling checker. *Communications of the ACM*, 28(5), 456–462.

Beutnagel, M., Conkie, A., Schroeter, J., Stylianou, Y., & Syrdal, A. (1999a). The AT&T Next-Gen TTS System. In *Joint Meeting of ASA, EAA, and DAGA* Berlin, Germany.

Beutnagel, M., Mohri, M., & Riley, M. (1999b). Rapid unit selection from a large speech corpus for concatenative speech synthesis. In *Proceedings of Eurospeech'99* Budapest.

Black, A. (2003). Unit Selection and Emotional Speech. In *Proceedings of Eurospeech'03*.

Black, A., Caley, R., & Taylor, P. (1999). The Festival speech synthesis system. http ://www.cstr.ed.ac.uk/projects/festival/.

Black, A. & Campbell, N. (1995). Optimising selection of units from speech databases for concatenative synthesis. In *Proceedings of Eurospeech'95* (pp. 581–584). Madrid, Spain.

Black, A. & Lenzo, K. (2001). Optimal Data Selection for Unit Selection Synthesis. In *Proceedings of 4rd ESCA/COCSADA Workshop on Speech Synthesis* (pp. 63–67).

Black, A., Taylor, P., & Caley, R. (1997). *The Festival Speech Synthesis System : System Documentation*. Technical report, University of Edinburgh.

Black, E., Jelinek, F., Lafferty, J., Magerman, D., Mercer, R., & Roukos, S. (1993). Towards History-Based Grammars : using richer models for probabilistic parsing. In *Meeting of the Association for Computational Linguistics* (pp. 31–37).

Boersma, P. (2003). Praat, a system for doing phonetics by computer. *Glot International*, 5(9–10), 341–345.

Borovikov, E., Zavorin, I., & Turner, M. (2004). A filter based post-OCR accuracy boost system. In *Proceedings of 1st ACM Workshop on Hardcopy Document Processing* (pp. 23–28).

Boîte, J., Dupont, S., Ris, C., Bataille, F., Deroo, O., Fontaine, V., & Zanoni, L. (1997). STRUT : Un logiciel complet pour l'entrainement et la reconnaissance de la parole. In *Proc. Premières Journées Scientifiques et Techniques FRANCIL* (pp. 41–44).

Boîte, R., Bourlard, H., Dutoit, T., Hancq, J., & Leich, H. (2000). *Traitement de la parole*. Lausanne : Presses Polytechniques et Universitaires Romandes.

Bozkurt, B., Dutoit, T., Prudon, R., D'Alessandro, C., & Pagel, V. (2004). Chapter 1 : Reducing discontinuities at synthesis time for corpus-based speech synthesis. In S. Narayanan & A. Alwan (Eds.), *Text To Speech Synthesis : New Paradigms and Advances*. Prentice Hall PTR.

Breen, A. & Jackson, P. (1998a). A phonologically motivated method of selecting non-uniform units. In *Proceedings of In ICSLP'98*.

Breen, A. & Jackson, P. (1998b). Non-Uniform Unit selection and the similarity metric within BT's laureate TTS system. In *Proceedings of 3rd ESCA/COCOSDA Workshop on Speech Synthesis* (pp. 201–206).

Breiman, L., Friedman, J., Stone, C., & Olshen, R. (1984). *Classification and Regression Trees*. Chapman & Hall/CRC.

Breslauer, D. (1995). The suffix tree of a tree and minimizing sequential transducers. In D. S. Hirschberg & E. W. Myers (Eds.), *Proceedings of the7th Annual Symposium on Combinatorial Pattern Matching* (pp. 116–129). Laguna Beach, CA : Springer-Verlag, Berlin.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing : A case study in part-of-speech tagging. *Computational Linguistics*, 21(4), 543–565.

Brill, E. & Moore, R. C. (2000). An improved error model for noisy channel spelling correction. In *Proceedings of Association for Computational Linguistics* (pp. 286–293). Morristown, NJ, USA : Association for Computational Linguistics.

Buchsbaum, A. & van Santen, J. (1996). Selecting training inputs via greedy rank covering. In *SODA '96 : Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms* (pp. 288–295). Philadelphia, PA, USA.

Bulyko, I. & Ostendorf, M. (2001). Unit selection for speech synthesis using splicing costs with weighted nite state transducers. In *Proceedings of Eurospeech'01.*

Bunke, H. (1995). Fast approximate matching of words against a dictionary. *Computing*, 55(1), 75–89.

Burges, C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2, 121–167.

Burr, D. (1987). Experiments with a connectionist text reader. In *IEEE International Conference on Neural Networks*, volume 4 (pp. 717–724).

Carter, D. (1992). Lattice-based word identification in CLARE. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics* (pp. 159–166).

Charniak, E. (1993). *Statistical Language Learning.* Cambride, MA : MIT Press.

Charpentier, F. & Stella, M. (1986). Diphone synthesis using an overlap-add technique for speech waveforms concatenation. In *Proceedings of IEEE'86* Tokyo, Japan.

Chen, S. & Goodman, J. (1998). *An Empirical Study of Smoothing Techniques for Language Modeling.* Technical Report 10-98, Computer Science Group, Harvard University.

Cherkassky, V. & Vassilas, N. (1989). Back-propagation networks for spelling correction. *Neural Networks*, 1(3), 166–173.

Choffrut, C. (1978). *Contribution à l'étude de quelques familles remarquables de fonctions rationnelles.* PhD thesis, Université Paris VII, Paris, France.

Chomsky, N. (1956). Three models for the description of language. *I.R.E. Transactions on Information Theory*, IT-2(3), 113–124.

Chomsky, N. & Halle, M. (1968). *The Sound Pattern of English*. New York, NY : Harper and Row.

Chomsky, N. & Schützenberger, M. (1963). The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, (pp. 118–161).

Church, K. & Gale, W. (1991). Probability scoring for spelling correction. *Statistics and Computing*, 1, 93–103.

Coker, C., Church, K., & Liberman, M. (1990). Morphology and rhyming : two powerful alternatives to letter-to-sound rules for speech synthesis. In *Proceedings of the Conference on Speech Synthesis* : European Speech Communication Association.

Collins, M. (1996). A New Statistical Parser Based on Bigram Lexical Dependencies. In *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics* (pp. 184–191). San Francisco : Morgan Kaufmann Publishers.

Colotte, V. & Beaufort, R. (2004). Synthèse vocale par sélection linguistiquement orientée d'unités non-uniformes : LiONS. In *Proceedings of JEP'04*.

Colotte, V. & Beaufort, R. (2005). Linguistic Features Weighting for a Text-to-Speech System Without Prosody Model. In *Proceedings of Interspeech'05* (pp. 2549–2552).

Comrie, B. (1996). *Language Universals and Linguistic Typology : Syntax and Morphology*. Blackwell. Seconde édition.

Cormen, T., Leiserson, C., & Rivest, R. (1990). *Introduction to algorithms*. Cambridge, MA, USA : MIT Press.

Cortes, C. & Vapnik, V. (1995). Support-Vector Networks. *Machine Learning*, 20(3), 273–297.

Crouzet, O. & Angoujard, J. (2006). Théorie de la syllabe et durées vocaliques : Vers une interprétation unifiée du rôle de la structure syllabique et de la nature des segments. In *Proceedings of JEP'06*.

Cucerzan, S. & Brill, E. (2004). Spelling correction as an iterative process that exploits the collective knowledge of web users. In *Proceedings of EMNLP 2004* (pp. 293–300).

Daciuk, J. & van Noord, G. (2001). A Finite-State Library for NLP. In *Proceedings of the 12th Meeting of Computational Linguistics in the Netherlands (CLIN 2001)* University of Twente, Enschede.

Daelemans, W. M. P. & van den Bosch, A. P. J. (1997). Language-independent data-oriented grapheme-to-phoneme conversion. In J. van Santen, R. Sproat, J. Olive, & J. Hirschberg (Eds.), *Progress in Speech Synthesis* (pp. 77–89). New York : Springer.

Damerau, F. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3), 171–176.

Davidson, L. (1962). Retrieval of misspelled names in an airline passenger record system. *Communications of the ACM*, 5, 169–171.

de la Briandais, R. (1959). File Searching Using Variable Length Keys. In *Proceedings of the Western Joint Computer Conference* (pp. 295–298).

Deerwester, S., Dumais, S., Landauer, T., Furnas, G., & Harshman, R. (1990). Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6), 391–407.

Deffner, R., Geiger, H., Kahler, R., Krempl, T., & Brauer, W. (1990). Recognizing words with connectionist architectures. In *Proceedings of INNC* (pp. 196). Paris, France.

Delattre, P. (1959). Rapports entre la durée vocalique, le timbre et la structure syllabique en francais. *The French Review*, 32(6), 547–552.

Dempster, A., Laird, N., & Rubin, D. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1), 1–38.

Deroo, O., Boîte, J., Ris, C., Fontaine, V., & Zanoni, L. (1997). Context Independent and Context Dependent Hybrid HMM/ANN Systems for Training Independent Tasks. In *Proceedings of Eurospeech'97* (pp. 1947–1950). Rhodes, Grèce.

Deroo, O., Leich, H., Boîte, J., Dupont, S., Ris, C., & Fontaine, V. (1996). Hybrid HMM/ANN systems for Speaker Independant Continuous Speech Recognition In French. In *Proceedings of ProRisc 8th Annual WorkShop on Circuits, System and Signal Processing* (pp. 137–141). Mierlo, The Netherlands.

Donovan, R. (2001). A New Distance Measure for Costing Spectral Discontinuities in Concatenative Speech Synthesisers. In *Proceedings of 4th ISCA Tutorial and Research Workshop on Speech Synthesis* (pp. 293–300). Atholl Palace Hotel, Scotland.

Draper, N. & Smith, H. (1998). *Applied Regression Analysis*. Wiley.

Du, M. & Chang, S. (1992). A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29, 281–302.

Dumais, S., Furnas, G., Landauer, T., Deerwester, S., & Harshman, R. (1988). Using Latent Semantic Analysis to Improve Access to Textual Information. In *Proceedings of CHI'88 : the Conference on Human Factors in Computing Systems*.

Dupont, S. & Ris (2003). Robust Feature Extraction and Acoustic Modeling at Multitel : Experiments on the Aurora Database. In *Proceedings of Eurospeech'03* Geneva.

Dupont, S., Ris, C., Couvreur, L., & Boîte, J. (2005). A study of implicit and explicit modeling of coarticulation and pronunciation variation. In *Proceedings of Interspeech'05*.

Dutoit, T. (1993). *High quality text-to-speech synthesis of the French language*. PhD thesis, TCTS Lab, Faculté Polytechnique de Mons, 31 bvd Dolez, B-7000 Mons, Belgium. 288 pages.

Dutoit, T., Bagein, M., Ruelle, A., Gilman, A., Malfrère, M., Mertens, P., & Pagel, V. (1998). Euler : Multilingual Text-To-Speech Project. In *Proceedings of Workshop on Text, Speech and Dialog* Brno.

Eckart, C. & Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1, 211–218.

Eilenberg, S. (1974). *Automata, Languages, and Machines*. Orlando, FL, USA : Academic Press Inc.

Eisner, J. (1996a). *An empirical comparison of probability models for dependency grammar*. Technical Report IRCS-96-11, IRCS, University of Pennsylvania.

Eisner, J. (1996b). Three New Probabilistic Models for Dependency Parsing : An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)* (pp. 340–345). Copenhagen.

Eisner, J. (2001). Expectation Semirings : Flexible EM for Finite-State Transducers. In *Proceedings of the ESSLLI Workshop on Finite-State Methods in Natural Language Processing (FSMNLP)*.

Eisner, J. (2002a). Parameter Estimation for Probabilistic Finite-State Transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 1–8). Philadelphia.

Eisner, J. (2002b). Transformational Priors Over Grammars. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 63–70). Philadelphia.

Eisner, J. (2003). Simpler and More General Minimization for Weighted Finite-State Automata. In *Proceedings of the Joint Meeting of the Human Language Technology Conference and the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)* (pp. 64–71). Edmonton.

Eisner, J. & Satta, G. (1999). Efficient Parsing for Bilexical Context-Free Grammars and Head-Automaton Grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 457–464). University of Maryland.

Elliott, R. (1988). *Annotating spelling list words with affixation classes.* Technical report, Bell Labs.

Fairon, C., Paumier, S., & Watrin, P. (2005). Can we parse without tagging ? In *Proceedings of the 2nd Language Technology Conference* (pp. 473–477). Poznan.

Fawcett, T. (2003). *ROC graphs : Notes and practical considerations for data mining researchers.* Technical Report HPL-2003-4, HP Laboratories, Palo Alto, CA, USA.

Finch, S. (1994). *Finite state automata library.* Documentation 0.01, LTG, University of Edinburgh.

Forgy, E. (1965). Cluster analysis of multivariate data : efficiency vs. interpretability of classifications. *Biometrics*, 21, 768–769.

Fredkin, E. (1960). Trie memory. In *Communications of the ACM* (pp. 490–499). Poznan.

Fredman, M. & Tarjan, R. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *ACM*, 34(3), 596–615.

Frishert, M. (2005). FIRE Works & FIRE Station. A Finite Automata & Regular Expression Playground. Master's thesis, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science.

Gale, W., Church, K., & Yarowsky, D. (1992). A Method for Disambiguating Word Senses in a Large Corpus. *Computers and the Humanities*, 26, 415–439.

Gale, W., Church, K., & Yarowsky, D. (1994). Discrimination Decisions for 100,000-Dimensional Spaces. In *Current Issues in Computational Linguistics : In Honour of Don Walker* (pp. 429–450). : Kluwer Academic Publishers.

Gaudissart, V., Ferreira, S., Mancas-Thillou, C., & Gosselin, B. (2005). Sypole : a Mobile Assistant for the Blind. In *Proceedings of EUSIPCO'05* Antalya, Turkey.

Golding, A. (1995). A Bayesian Hybrid Method for Context-Sensitive Spelling Correction. In *Proceedings of the Third Workshop on Very Large Corpora* (pp. 39–53). Somerset, New Jersey : Association for Computational Linguistics.

Golding, A. & Roth, D. (1996). Applying Winnow to Context-Sensitive Spelling Correction. In *Proceedings of International Conference on Machine Learning* (pp. 182–190).

Golding, A. & Roth, D. (1999). A winnow-based approach to context-sensitive spelling correction. *Machine Learning*, 34(1–3), 107–130.

Golding, A. & Schabes, Y. (1996). Combining trigram-based and feature-based methods for context-sensitive spelling correction. In A. Joshi & M. Palmer (Eds.), *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics* (pp. 71–78). San Francisco : Morgan Kaufmann Publishers Inc.

Goshtasby, A. & Ehrich, R. (1988). Contextual word recognition using probabilistic relaxation labeling. *Pattern Recognition*, 21(5), 455–462.

Granger, R. (1983). The NOMAD system : Expectation-based detection and correction of syntactically and semantically ill-formed text. *American Journal of Computational Linguistics*, 9(3-4), 188–196.

Gray, R. (1984). Vector Quantization. *IEEE ASSP Magazine*, (pp. 4–29).

Gross, M. (1984). Lexicon-grammar and the syntactic analysis of French. *Coling'84*, (pp. 275–282).

Gross, M. & Perrin, D. (1989). Electronic Dictionnaries and Automata in Computational Linguistics. *Lecture Notes in Computer Science*, 377.

Gunter, C. & Mitchell, J. (1994). *Theoretical Aspects of Object-Oriented Programming : Types, Semantics, and Language Design*. MIT Press.

György, G. & Csaba, P. (1994). Lexical processing in an agglutinative language and the organization of the lexicon. *Folia Linguistica*, 28, 175–204.

Hanneforth, T. (2006). *Finite-State Machines - Theory and Applications*. Technical Report 29/11/2006, Universität Potsdam.

Hanson, A., Riseman, E., & Fisher, E. (1976). Context in word recognition. *Pattern Recognition*, 8, 35–45.

Harmon, L. (1972). Automatic recognition of print and script. In *IEEE 60* (pp. 1165–1176).

Harris, M. & Vincent, N., Eds. (1990). *The Romance Languages*. Oxford University Press.

Harris, Z. (1962). *String analysis of sentence structure*. The Hague : Mouton co.

Harrison, M. (1978). *Introduction to Formal Language Theory*. Boston, MA, USA : Addison-Wesley.

Hirst, G. & Budanitsky, A. (2005). Correcting Real-Word Spelling Errors by Restoring Lexical Cohesion. *Computational Linguistics*, 11(1), 87–111.

Ho, T., Hull, J., & Srihari, S. (1991). Word recognition with multi-level contextual knowledge. In *IDCAR-91* (pp. 905–915).

Holub, J. & Zdárek, J., Eds. (2007). *Implementation and Application of Automata, 12th International Conference (CIAA 2007)*. Prague, Czech Republic : Springer. Revised Selected Papers.

Hopcroft, J., Motwani, R., & Ullman, J., Eds. (1979). *Introduction to Automata Theory, Languages, and Computation*. Massachusetts : Addison-Wesley.

Hull, J. (1987). Hypothesis testing in a computational theory of visual word recognition. In *AAAI-87, 6th national conference on Artificial Intelligence* (pp. 718–722).

Hunt, A. & Black, A. (1996). Unit selection in a concatenative speech synthesis system using a large speech database. In *Proceedings of ICASSP'96*, volume 1 (pp. 373–376). Atlanta, Georgia.

Janda, R. & Joseph, B., Eds. (2004). *The Handbook of Historical Linguistics*. Blackwell.

Jelinek, F. & Mercer, R. (1980). Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of Pattern Recognition in Practice* (pp. 381–397).

Johnson, C. (1972). *Formal aspects of phonological description*. The Hague : Mouton.

Johnson, M. (1998a). Finite-state Approximation of Constraint-based Grammars using Left-corner Grammar Transforms. In *COLING-ACL* (pp. 619–623).

Johnson, M. (1998b). PCFG models of linguistic tree representations. *Computational Linguistics*, 24(4), 613–632.

Jones, M. & Martin, J. (1997). Contextual spelling correction using latent semantic analysis. In *Proceedings of the fifth conference on Applied natural language processing* (pp. 166–173). San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

Jones, M., Story, G., & Ballard, B. (1991). Integrating multiple knowledge sources in a bayesian OCR post-processor. In *IDCAR-91* (pp. 925–933).

Kahan, S., Pavlidis, T., & Baird, H. (1987). On the recoginition of printed characters of any font and size. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 9(3), 274–288.

Kanthak, S. & Ney, H. (2004). FSA : An Efficient and Flexible C++ Toolkit for Finite State Automata Using On-Demand Computation. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004)* (pp. 510–517). Barcelona, Spain.

Kaplan, R. & Kay, M. (1994). Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20(3), 331–378.

Karttunen, L. (1994). Constructing lexical transducers. In *Proceedings of COLING'94* Kyoto, Japan.

Karttunen, L. (1995). The replace operator. In *Meeting of the Association for Computational Linguistics* (pp. 16–23).

Karttunen, L., Chanod, J., Grefenstette, G., & Schiller, A. (1996). Regular expressions for language engineering. *Natural Language Engineering*, 2(4), 305–238.

Karttunen, L., Kaplan, R., & Zaenen, A. (1992). Two-level morphology with composition. In *Proceedings of COLING'92* (pp. 141–148). Nantes, France.

Keller, F. & Lapata, M. (2003). Using the Web to obtain frequencies for unseen bigrams. *Computational Linguistics*, 29(3), 459–484.

Kempe, A. (2000). Part-of-Speech tagging with two sequential transducers. In *Proceedings of CLIN 2000* (pp. 88–96).

Kempe, A., Baeijs, C., Gaál, T., Guingne, F., & Nicart, F. (2007). WFSC – A New Weighted Finite State Compiler. In *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA 2007)* (pp. 108–119).

Kernighan, M. (1991). *Specialized Spelling Correction for a TDD system.* Technical report, AT&T Bell Labs.

Kilgarriff, A. & Grefenstette, G. (2003). Special Issue on Web as Corpus. *Computational Linguistics*, 29(3).

Klabbers, E. & Veldhuis, R. (1998). On the Reduction of Concatenation Artefacts in Diphone Synthesis. In *Proceedings of ICSLP'98* Sydney.

Klatt, D. (1980). Software for a cascade/parallel formant synthesizer. *The Journal of the Acoustical Society of America*, 67(3), 971–995.

Kleene, S. (1956). Representation of events in nerve nets and finite automata. In *Automata Studies.*

Klinkenberg, J.-M. (1994). *Des langues romanes.* Louvain-la-Neuve : Editions Duculot. 2^e édition.

Kneser, R. & Ney, H. (1995). Improved backing-off for m-gram language modeling. In *Proceedings of ICASSP'95*, volume 1 (pp. 181–184).

Knuth, D., Ed. (1973). *The art of programming*, volume 3 : Sorting and searching. Mass : Addison-Wesley.

Kohavi, R. (1995). A study of Cross-Validation and Bootstrap for Accuracy Estima-tion and Model Selection. In *Proceedings of IJCAI'95* (pp. 1137–1143).

Kolak, O., Byrne, W., & Resnik, P. (2003). A generative probabilistic OCR model for NLP applications. In *Proceedings of NAACL-Human Language Technology*, volume 1 (pp. 55–62).

Kominek, J. & Black, A. (2004). The CMU Arctic speech databases. In *Proceedings of 5th ISCA Speech Synthesis Workshop* (pp. 223–224). Pittsburgh, PA., USA.

Koskenniemi, K. (1983). *Two-level Morphology : A General Computational Model for Word-Form Recognition and Production.* Technical report, Department of General Linguistics, University of Helsinki.

Kuich, W. & Salomaa, A. (1986). *Semirings, automata, languages*, volume 5 of *EATCS Monographs on Theoretical Computer Science.* Berlin, Germany : Springer-Verlag.

Kukich, K. (1988). Variations on a backpropagation name recognition net. In *Proceedings of the Advanced Technology Conference*, volume 2 (pp. 722–735). Washington D.C.

Kukich, K. (1990). A comparison of some novel and traditional lexical distance metrics for spelling correction. In *INNC-90* (pp. 309–313).

Kukich, K. (1992a). Spelling correction for the telecommunications network for the deaf. *Communications of the ACM*, 35(5), 80–90.

Kukich, K. (1992b). Techniques for automatically correcting words in text. In *ACM Computing Surveys*, volume 24(4) (pp. 377–439).

Kullback, S. & Leibler, R. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, 22, 79–86.

Kupiec, J. (1992). Robust Part-of-Speech tagging using a Hidden Markov Model. *Computer Speech and Language*, 6, 225–242.

Kučera, H. & Francis, W. (1967). *Computational Analysis of Present-Day American English.* Providence, RI : Brown University Press.

Lamel, L., Gauvain, J., & Eskénazi, M. (1991). BREF, a large vocabulary spoken corpus for French. In *Proceedings of Eurospeech'91* (pp. 505–508).

Lass, R. (1997). *Historical linguistics and language change.* Cambridge University Press.

Lazarov, M. (2006). Finite-State Methods for Spelling Correction. BA thesis, Seminar für Sprachwissenschaft, Dept. of Linguistics, University of Tübingen.

Lee, Y., Evens, M., Michael, J., & Rovick, A. (1990). *Spelling correction for an intelligent tutoring system.* Technical report, Department of Computer Science, Illinois Institue of Technology, Chicago.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics*, 10, 707–710.

Liberman, A., Ingemann, F., Lisker, L., Delattre, P., & Cooper, F. (1959). Minimal rules for synthesizing speech. *Journal of the Acoustical Society of America*, 31, 1490–1499.

Liberman, M. & Church, K. (1991). Text Analysis and Word Pronunciation in Text-to-Speech Synthesis. In S. Furui & M. Sondhi (Eds.), *Advances in Speech Signal Processing* (pp. 791–831). Dekker.

Lidstone, G. (1920). Note on the general case of the Bayes-Laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries*, 8, 182–192.

Linde, Y., Buzo, A., & Gray, R. (1980). An Algorithm for Vector Quantizer Design. *IEEE Transactions on Communications*, (pp. 702–710).

littlestone, N. (1988). Learning Quickly When Irrelevant Attributes Abound : A New Linear-Threshold Algorithm. *Mach. Learn.*, 2(4), 285–318.

Littlestone, N. & Warmuth, M. (1994). The weighted majority algorithm. *Information and Computation*, 108(2), 212–261.

Liu, J. (2000). *Real-time Systems*. Prentice Hall.

Lopresti, D. & Zhou, J. (1997). Using consensus sequence voting to correct OCR errors. *Computer Vision and Image Understanding*, 67(1), 39–47.

Lucas, S., Panaretos, A., Sosa, L., Tang, A., Wong, S., & Young, R. (2003). *Robust reading competition*. Washington, DC, USA.

Malfrère, F., Dutoit, T., & Mertens, P. (1990). Un générateur de prosodie « tout automatique ». In *JEP 98* (pp. 147–150).

Malfrère, F., Dutoit, T., & Mertens, P. (1998). Fully Automatic Prosody Generator for Text-to-Speech Synthesis. In *Proceedings of ICSLP'98* (pp. 1395–1398). Sidney, Australia.

Mancas-Thillou, C. (2006). *Natural Scene Text Understanding*. PhD thesis, FPMS, Mons, Belgium.

Mangu, L. & Brill, E. (1997). Automatic rule acquisition for spelling correction. In *Proceedings of 14th International Conference on Machine Learning* (pp. 187–194). : Morgan Kaufmann Publishers Inc.

Mansfield, H. (1993). *Machiavel. L'art de la guerre*. Flammarion.

Marcus, M., Santorini, B., & Marcinkiewicz, M. (1993). Building a large annotated corpus of English : The Penn Treebank. *Computational Linguistics*, 19(2), 313–330.

Markel, J. & Gray, A. (1980). *Linear Prediction of speech.* New York : Springer Verlag.

Mays, E., Damerau, E., & Mercer, R. (1991). Context-based spelling correction. *Information Processing and Management*, 27(5), 517–522.

McCoy, K. (1991). Generating context-sensitive responses to object-related misconceptions. *Artificial Intelligence*, 41, 157–195.

McNaughton, R. & Yamada, H. (1960). Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers EC*, 9(1), 39–47.

McQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1 (pp. 281–297).

Mealy, G. (1955). A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5), 1045–1079.

Means, L. (1988). Cn yur cmputr raed ths ? In *Proceedings of the 2nd Applied Natural Language Processing conference* (pp. 93–100).

Mertens, P. (2001). Vertex, a chart parser for unification grammars.

Miki, B. & Mchugh, S. (2004). Selectable marker genes in transgenic plants : applications, alternatives and biosafety. *Journal of Biotechnology*, 107(3), 193–232.

Minton, S., Hayes, P., & Fain, J. (1985). Controlling search in flexible parsing. In *Proceedings of the 9th IJCAI* (pp. 785–787).

Mitton, R. (1987). Spelling checkers, spelling correctors and the misspellings of poor spellers. *Information Processing and Managment*, 23(5), 495–505.

Mohri, M. (1994a). Compact representations by finite-state transducers. In *Meeting of the Association for Computational Linguistics* (pp. 204–209).

Mohri, M. (1994b). Syntactic Analysis by Local Grammars Automata : an Efficient Algorithm. In *Proceedings of COMPLEX'94*.

Mohri, M. (1996). On some Applications of Finite-State Automata Theory to Natural Language Processing. *Journal of Natural Language Engineering*, 2, 1–20.

Mohri, M. (1997a). Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2), 269–311.

Mohri, M. (1997b). On the use of sequential transducers in Natural Language Processing. In E. Roche & Y. Schabes (Eds.), *Finite-State Language Processing, chapter 12* (pp. 355–382). Cambridge, Massachusetts : MIT Press.

Mohri, M. (2000). Minimization Algorithms for Sequential Transducers. *Theoretical Computer Science*, 234, 177–201.

Mohri, M. (2001). Weighted Grammar Tools : the GRM Library. In J.-C. Junqua & G. van Noord (Eds.), *Robustness in Language and Speech Technology* (pp. 165–186). Kluwer Academic Publishers, The Netherlands.

Mohri, M. (2002a). Generic ϵ-removal and input ϵ-normalization algorithms for weighted transducers. *International Journal of Foundations of Computer Science*, 13(1), 129–143.

Mohri, M. (2002b). Semiring Frameworks and Algorithms for Shortest-Distance Problems. *Journal of Automata, Languages and Combinatorics*, 7(3), 321–350.

Mohri, M. (2003). Edit-Distance of Weighted Automata. *Lecture Notes in Computer Science*, 2608, 1–23.

Mohri, M. & Nederhof, M.-J. (2001). Regular approximation of context-free grammars through transformation. In J.-C. Junqua & G. van Noord (Eds.), *Robustness in Language and Speech Technology, chapter 9* (pp. 153–163). : Kluwer Academic Publishers.

Mohri, M., Pereira, F., & Riley, M. (1996). Weighted automata in text and speech processing. In *ECAI'96 Workshop* (pp. 46–50).

Mohri, M., Pereira, F., & Riley, M. (1997). A Rational Design for a Weighted Finite-State Transducer Library. In *Workshop on Implementing Automata* (pp. 144–158).

Mohri, M., Pereira, F., & Riley, M. (2000). The Design Principles of a Weighted Finite-State Transducer Library. *Theoretical Computer Science*, 231(1), 17–32.

Mohri, M., Pereira, F., & Riley, M. (2001). Generic ϵ-Removal Algorithm for Weighted Automata. *Lecture Notes in Computer Science*, 2088, 230–242.

Mohri, M., Pereira, F., & Riley, M. (2002). Weighted finite state transducers in speech recognition. *Computer Speech and Language*, 16(1), 69–88.

Mohri, M. & Riley, M. (1999). Network optimizations for large-vocabulary speech recognition. *Speech Communication*, 28, 1–12.

Mohri, M. & Riley, M. (2002). An efficient algorithm for the n-best-strings problem. In *Proceedings of ICSLP'02*.

Mohri, M. & Sproat, R. (1996). An efficient compiler for weighted rewrite rules. In *Meeting of the Association for Computational Linguistics* (pp. 231–238).

Moore, E. (1956). Gedanken-Experiments on sequential machines. *Automata Studies*, (pp. 129–153).

Moulines, E. & Charpentier, F. (1990). Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones. *Speech Communication*, 9, 453–467.

Moulines, E. & Laroche, J. (1995). Non-parametric techniques for pitch-scale and time-scale modification of speech. *Speech Communication*, 16, 175–206.

Myers, C. & Rabiner, L. (1981). A comparative study of several Dynamic Time-Warping algorithms for connected word recognition. *The Bell System Technical Journal*, 60((7)), 1389–1409.

Naughton, P. (1996). *The Java Handbook*. Mcgraw-Hill Osborne Media.

Nederhof, M.-J. (2000a). Practical Experiments with Regular Approximation of Context-Free Languages. In *Computational Linguistics*, volume 26 (pp. 17–44). MIT Press for the Association for Computational Linguistics.

Nederhof, M.-J. (2000b). Regular approximation of CFLs : a grammatical view. In H. Bunt & A. Nijholt (Eds.), *Advances in Probabilistic and other Parsing Technologies, chapter 12* : Kluwer Academic Publishers.

Neuhoff, D. (1975). The Viterbi algorithm as an aid in text recognition. *IEEE Trans. Information Theory*, 21, 222–226.

Ney, H., Essen, U., & Kneser, R. (1994). On structuring probabilistic dependences in stochastic language modelling. *Computer, Speech and Language*, 8, 1–38.

Nérode, A. (1958). Linear Automaton Transformations. In *Proceedings of the American Mathematical Society*, volume 9 (pp. 541–544).

Oflazer, K. (1996). Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1), 73–89.

Paulo, S. & Oliveira, L. (2004). Automatic phonetic alignment and its confidence measures. In *Proceedings of ESTAL04*.

Pereira, F. & Riley, M. (1997). Speech Recognition by Composition of Weighted Finite Automata. In *Finite-State Language Processing, chapter 15* (pp. 431–453).

Pereira, F. & Wright, R. (1997). Finite-state approximation of phrase structure grammars. In E. Roche & Y. Schabes (Eds.), *Finite-State Language Processing, chapter 5* (pp. 149–173). Cambridge, Massachusetts : MIT Press.

Pereira, F. Riley, M. & Sproat, R. (1994). Weighted rational transductions and their application to human language processing. In *ARPA Workshop on Human Language Technology*.

Peterson, J. (1980). Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12), 676–684.

Peterson, J. (1986). A note on undetected typing errors. *Communications of the ACM*, 29(7), 633–637.

Pierret, J. (1994). *Phonétique historique du français et notions de phonétique générale*. Série pédagogique de l'Institut de linguistique de Louvain. Louvain-la-Neuve, Nouvelle édition : Peeters.

Piskorski, J. (2002). *Finite-State Machine Toolkit*. Technical Report RR-02-04, DFKI, Saarbrücken.

Pitrat, J. (1983). *Réalisation d'un analyseur lexicographique général*. Technical report, Institut de programmation, Paris VI.

Pollock, J. & Zamora, A. (1983). Collection and characterization of spelling errors in scientific and scholarly text. *Journal of the American Society for Information Science*, 34(1), 51–58.

Pollock, J. & Zamora, A. (1984). Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, 27(4), 358–368.

Pols, L. (1990). Does improved performance of a rule synthesizer also contribute to more phonetic knowledge ? In *Proceedings of the ESCA tutorial day on speech synthesis* (pp. 50–54).

Prudon, R. & d'Alessandro, C. (2001). A selection/concatenation tts synthesis system : databases developement, system design, comparative evaluation. In *ISCA/IEEE 4th Tutorial and Research Workshop on Speech Synthesis* (pp. 201–206). Pitlochry, Schotland.

Pulman, S. (1986). Grammars, parsers, and memory limitations. *Language and Cognitive Processes*, 1(3), 197–225.

Quinlan, J. (1979). Discovering rules by induction from large collections of examples. In D. Michie (Ed.), *Expert systems in the microelectronic age*. Edinburg University Press.

Quinlan, J. (1986). Induction of decision trees. *Machine learning*, 1, 81–106.

Quinlan, J. (1993). *C4.5 : programs for machine learning*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

Raymond, D. & Wood, D. (1995). Grail : A c++ library for automata and expressions. *Journal of Symbolic Computation*, 17, 341–350.

Rey, A. (1995). *Essays on Terminology*. Amsterdam : John Benjamins.

Ringlstetter, C., Hadersbeck, M., Schulz, K., & Mihov, S. (2007). Text Correction Using Domain Dependent Bigram Models from Web Crawls. In *Workshop on Analytics for Noisy Unstructured Text Data (IJCAI-2007)* (pp. 47–54). Hyderabad, India.

Riseman, E. & Hanson, A. (1974). A contextual postprocessing system for error correction using binary n-grams. In *IEEE Transactions on Computers*, volume 23 (pp. 480–493).

Roche, E. (1993). *Analyse Syntaxique Transformationnelle du Français par Transducleurs el Lexique-Grammaire*. PhD thesis, Université Paris 7, Paris, France.

Roche, E. & Schabes, Y., Eds. (1997). *Finite-State Language Processing*. Cambridge, Massachusetts : MIT Press.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In D. Rumelhart & J. McClelland (Eds.), *Parallel Distributed Processing : Explorations in the Microstructure of Cognition*. Bradford books/MIT Press.

Sabah, G. (1988). *L'intelligence artificielle et le langage. I. Représentation des connaissances*. Paris : Hermès.

Sagisaka, Y. (1988). Speech synthesis by rule using an optimal selection of non-uniform synthesis units. In *Proceedings of ICASSP'88*, volume 1 (pp. 679–682).

Sagisaka, Y. (1992). The ATR ν-Talk Speech Synthesis System. In *Proceedings of ICSLP'92*, volume 1 (pp. 483–486).

Saraiva, S. (2002). HaLeX : A Haskell Library to Model, Manipulate and Animate Regular Languages. In *Proceedings of the ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02)* Pittsburgh, USA.

Schaback, J. & Li, F. (2007). Multi-Level Feature Extraction for Spelling Correction. In *Proceedings of workshop on Analytics for Noisy Unstructured Text Data (IJCAI-2007)* (pp. 79–86). Hyderabad, India.

Schank, R., Lebowitz, M., & Birnbaum, L. (1980). An integrated understander. *American Journal of Computational Linguistics*, 6(1), 13–30.

Schmid, H. (2005). *SFST Manual*. Institute for Natural Language Processing, University of Stuttgart.

Schröder, M. (2003). Emotional speech synthesis for emotionally-rich virtual worlds. In *Proceedings of Workshop on emotionally rich virtual worlds with emotion synthesis at the 8th International Conference on 3D Web Technology*.

Schützenberger, M. (1961). On the definition of a family of automata. *Information and Control*, 4, 245–270.

Schützenberger, M. (1977). Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4(1), 47–57.

Schulz, K. & Mihov, S. (2002). Fast String Correction with Levenshtein-Automata. *International Journal of Document Analysis and Recognition*, 5(1), 67–85.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27, 379–423.

Silberztein, M. (1993). *Dictionnaires électroniques et analyse automatique de textes. Le système INTEX.* Paris : Masson.

Silberztein, M. (v.412). *INTEX.*

Sitar, E. (1961). *Machine recognition of cursive script. The use of context for error detection and correction.* Technical report, Bell Labs.

Sjölander, K. (2001). Automatic alignment of phonetic segments.

Stevens, K. (1990). Control parameters for synthesis by rule. In *Proceedings of the ESCA tutorial day on speech synthesis* (pp. 27–37).

Stevens, S., Volkmann, J., & Newmann, E. (1937). A scale for the measurement of the psychological magnitude of pitch. *Journal of the Acoustical Society of America*, 8(185).

Stolcke, A. (2002). SRILM – an extensible language modeling toolkit.

Strohmaier, C., Ringlstetter, C., Schulz, K. U., & Mihov, S. (2003). Lexical postcorrection of OCR-results : The web as a dynamic secondary dictionary ? In *Proceedings of ICDAR'03* (pp. 1133–1137).

Stroustrup, B. (2000). *The C++ Programming Language (Special Edition).* USA : Addison Wesley.

Suri, L. (1991). *Language transfer : A foundation for correcting the written English of ASL signers.* Technical Report 91-19, Department of Computer and Information Sciences, University of Delaware.

Suri, L. & McCoy, K. (1991). *Language transfer in deaf writing : A correction methodology for an instructional system.* Technical Report 91-20, Department of Computer and Information Sciences, University of Delaware.

Tapanainen, P. & Jarvinen, T. (1994). Syntactic analysis of natural language using linguistic rules and corpus-based patterns. In *Proceedings of Computational Linguistics* (pp. 629–634). Kyoto.

Tarjan, R. & Yao, A. (1979). Storing a sparse table. *Communications of the ACM*, 22(11), 606–611.

Taylor, P. (2000). Concept-to-Speech synthesis by phonological structure matching. *Philosophical Transactions of the Royal Society, Series A*, 356(1769), 1403–1416.

Taylor, P. & Black, A. (1997). Automatically Clustering Similar Units for Unit Selection in Speech Synthesis. In *Proceedings of Eurospeech'97* (pp. 601–604). Rhodes, Greece.

Taylor, P. & Black, A. (1999). Speech synthesis by phonological structure matching. In *Proceedings of Eurospeech'99* (pp. 1531–1534). Budapest, Hungary.

Thillou, C., Ferreira, S., & Gosselin, B. (2005). An embedded application for degraded text recognition. *Eurasip Jour. on Applied Signal Processing*, 13, 2127–2135.

Toutanova, K. & Moore, R. C. (2002). Pronunciation Modeling for Improved Spelling Correction. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics* (pp. 144–151).

Trawick, D. (1983). *Robust Sentence Analysis and Habitability.* PhD thesis, California Institute of Technology, Pasadena, California.

Tsao, Y. (1990). A lexical study of sentences typed by hearing-impaired TDD users. In *13th International Symposium on Human Factors in Telecommunications* (pp. 197–201).

Turba, T. (1981). Checking for spelling and typographical errors in computer-based text. In *ACM SIGPLAN/SIGOA Symposium on Text Manipulation* (pp. 51–60).

Turing, A. (1950). Computing Machinery and Intelligence. *Mind*, 49, 433–460.

Tzoukermann, E. & Radev, D. (1996). Using Word Class for Part-of-Speech disambiguation. In *4th Workshop on Very Large Corpora* (pp. 1–13).

Ullian, J. (1967). Partial Algorithm Problems for Context Free Languages. *Information and Control*, 11, 80–101.

Ullmann, J. (1977). A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 20(2), 141–147.

Valiant, L. (1995). Rationality. In *Proceedings of CoLT'95 : workshop on Computational Learning Theory* : Morgan Kaufmann Publishers Inc.

Valiant, L. (2004). *Circuits of the Mind.* Oxford : Oxford University Press.

van den Bosch, A. (1997). *Learning to Pronounce Written Words : A Study in Inductive Language Learning.* PhD thesis, University of Maastricht, Maastricht, The Netherlands.

van Santen, J. & Buchsbaum, A. (1997). Methods for Optimal Text Selection. In *Proceedings of Eurospeech'97* (pp. 553–556). Rhodes, Greece.

Vine, D. & Sahandi, R. (2000). Synthesising Emotional Speech by Concatenating Multiple Pitch Recorded Speech Units. In *Proceedings of ISCA Workshop on Speech and Emotion*.

Volk, M. (2001). Exploiting the www as a corpus to resolve pp attachment ambiguities. In *Proceedings of Corpus Linguistics*.

Voorhees, E. & Buckland, L., Eds. (2006). *Fifteenth Text REtrieval Conference (TREC 2006)*, Gaithersburg, Maryland.

Voutilainen, A. (2001). Tagging and parsing with rules : the case of Swedish. *Lingvisticæ Investigationes*, 24(1), 43–66.

Voyles, J. (1992). *Early Germanic Grammar : Pre-, Proto-, and Post-Germanic Languages*. Academic Press.

Wagner, R. (1974). Order-n correction for regular languages. *Communications of the ACM*, 17(5), 265–268.

Wagner, R. & Fisher, M. (1974). The string to string correction problem. *Journal of the Association for Computing Machinery (ACM)*, 21(1), 168–173.

Waltz, D. (1978). An English language question answering system for a large rational database. *Communications of the ACM*, 21(7), 526–539.

Watson, B. (1994a). *An Introduction to the FIRE Engine : A C++ Toolkit for FInite Automata and Regular Expressions*. Computing Science Report 94/21, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

Watson, B. (1994b). *The Design of the FIRE Engine : A C++ Toolkit for FInite Automata and Regular Expressions*. Computing Science Report 94/22, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

Watson, B. (1999). The OpenFIRE Initiative. In *Proceedings of the International Conference on Computer Processing of Oriental Languages*, volume 2 (pp. 421–424). Tokushima, Japan.

Williams, H. & Zobel, J. (2005). Searchable words on the web. *International Journal of Digital Libraries*, 5(2), 99–105.

Winograd, T. (1972). *Understanding Natural Language*. Orlando, FL, USA : Academic Press, Inc.

Witten, I. & Bell, T. (1991). The zero-frequency problem : estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 34(4), 1085–1094.

Yannakoudakis, E. & Fawthrop, D. (1983). The rules of spelling errors. *Information Processing and Managment*, 19(2), 87–99.

Yarowsky, D. (1994). Decision Lists for Lexical Ambiguity Resolution : Application to Accent Restoration in Spanish and French. In *Meeting of the Association for Computational Linguistics* (pp. 88–95).

Yi, J., Glass, J., & Hetherington, I. (2000). A Flexible, Scalable Finite-State Transducer Architecture for Corpus-Based Concatenative Speech Synthesis.

Yli-Jyrä, A., Karttunen, L., & Karhumäki, J., Eds. (2005). *Finite-State Methods and Natural Language Processing, International Workshop, FSMNLP 2005*. University of Helsinki, Finland : Springer. Revised Papers.

Zhang, J., Chen, X., Hanneman, A., Yang, J., & Waibel, A. (2002). A Robust Approach for Recognition of Text Embedded in Natural Scenes. In *Proceedings of Int. Conf. on Pattern Recognition*.

Zhu, X. & Rosenfeld, R. (2001). Improving trigram language modeling with the world wide web. In *Proceedings of Int. Conf. on Acoustics, Speech, and Signal Processing*.

Annexes

Annexe A

Bibliothèques et outils à états finis

La Table A.1 présente une liste des bibliothèques et outils à états finis disponibles sur internet. Cette liste n'est certainement pas exhaustive. Elle reprend par contre les bibliothèques et les outils les plus complets et les plus utilisés, et quelques systèmes en cours de développement qui nous semblent prometteurs. Le lecteur intéressé pourra compléter la liste proposée en visitant la page du site KitWiki consacrée aux machines à états finis [1], ou en consultant les actes des conférences FSMNLP (Yli-Jyrä *et al.* 2005) et CIAA (Holub & Zdárek 2007).

Système	Section	Système	Section	Système	Section
Amore	1.1	HaLeX	1.1	REGI	2
ASTL	1.1	Hopskip	2	RWTH FSA	1.3
AttiasFSTools	1.2	INTEX	2	SFST	2
Carmel	1.3	Jacaranda	2	Speech Tools Library	2
DFKIFSMToolkit	1.3	LASH	2	SRILM	2
DK.BRICS	1.1	LEXC	2	Statechart	2
Edinburgh FSA	1.1	Lextools	2	UCFSM	1.3
Fadd	2	Lintouch	1.1	Unitex	2
FIRE	1.1	MAP-3.1	2	Vaucanson	1.3
FSM Library	1.3	MIT FST	1.3	WFSC	1.3
Gdansk FSA/UTR	1.2	OMAC FSM Library	1.2	WFST	1.3
GFSM	1.3	openFst	1.3	XFST	1.2
Grail	1.1	PC-KIMMO	2	XFST2FSA	2
GRM	2	PFST	1.3		
Groningen FSA	1.3	REGAL	1.1		

TAB. A.1: Classement alphabétique des bibliothèques et outils à états finis. Le numéro de section indique la section où le système est décrit

[1] KitWiki : forums.csc.fi/kitwiki/pilot/view/KitWiki/FsmReg.

La Section 1 liste uniquement les bibliothèques, tandis que la Section 2 rassemble, sous l'appellation *outils*, les environnements, compilateurs et interpréteurs disponibles. Dans chaque section, les différents systèmes sont classés par ordre alphabétique.

En Section 1, nous avons réparti les différentes bibliothèques disponibles entre celles qui gèrent uniquement des automates classiques (Section 1.1), des machines classiques (Section 1.2), et des machines classiques et pondérées (Section 1.3).

Note 0.1. Certains systèmes acceptent le formalisme de la morphologie à deux niveaux. Il s'agit d'un modèle proposé par Koskenniemi (1983), dans lequel un mot est représenté comme une correspondance entre sa forme lexicale et sa forme de surface. Pour désigner un modèle ou une règle de ce type, nous parlons de *modèle* ou de *règle de Koskenniemi*. Pour une description plus complète, nous renvoyons le lecteur intéressé à la référence citée.

1 Les bibliothèques

1.1 Automates classiques

Amore. Ensemble d'algorithmes permettant la manipulation d'automates et d'expressions régulières. La bibliothèque tourne sous Linux.
Site : amore.sourceforge.net/.

ASTL. Bibliothèque développée en C++, qui utilise massivement le standard STL [2]. Ce standard définit des conteneurs génériques pour C++, qui facilitent considérablement le développement, mais ne sont pas bien gérés par les compilateurs pour plateforme embarquée. En outre, le caractère générique des conteneurs STL, s'il facilite le développement, nuit à l'efficacité du code.
Site : www-igm.univ-mlv.fr/~lemaout/ASTL/DOC/index.html.

DK.BRICS. Cette bibliothèque supporte les opérations classiques (concaténation, union, étoile de Kleene) et quelques autres (intersection, complément, etc.). Elle est écrite en Java et représente les alphabets en Unicode UTF-16. Il est possible de télécharger le code source ou une version pré-compilée.
Site : www.brics.dk/automaton/.

Edinburgh FSA. Bibliothèque C++ compilable sous g++, qui implémente des automates déterministes. Elle définit les opérateurs réguliers et permet donc la construction incrémentale d'automates. Elle autorise également la compilation de strings qui représentent des expressions régulières (Finch 1994).
Site : nl.ijs.si/et/Thesis/ALE-RA/.

FIRE. Bibliothèque écrite en C++ (version 3.0) par Watson (1994a,b). Elle permet la manipulation d'automates et la conversion d'expressions régulières en automates. Des opérations comme le complément et l'intersection n'y figurent cependant pas encore.

[2] *Standard Template Library*. Voir www.sgi.com/tech/stl/.

La bibliothèque a été développée avec l'objectif d'être efficace, et donc utilisable dans des applications réelles. Plusieurs versions de FIRE coexistent :

- FIRE Engine, qui en est la base et contient plusieurs versions des algorithmes de construction et de minimisation.
- FIRE Lite, qui est une version allégée de FIRE Engine.
- FIRE Works et FIRE Station, un environnement graphique permettant de manipuler les automates et les expressions régulières (Frishert 2005).

Cet ensemble d'outils ne doit pas être confondu avec l'initiative *OpenFIRE*, initiée par Watson (1999). L'objectif d'openFIRE est de poser les bases d'un outil à états finis unifié, qui pourrait accueillir les algorithmes proposés par les spécialistes du domaine.
Site : `www.cs.sunysb.edu/~algorith/implement/watson/implement.shtml`.
FTP : `ftp.win.tue.nl/pub/techreports/pi/automata/toolkit/v1.1/fire.tar.gz`.

Grail. Bibliothèque écrite en C++, dont les fonctionnalités, décrites dans (Raymond & Wood 1995), sont un peu plus complètes que celles de DK.BRICS. Le code source et différents binaires peuvent être téléchargés.
Site : `www.cse.ust.hk/~dwood/grail/index.html`.

HaLeX. Bibliothèque, écrite en Haskell [3], qui a principalement été définie pour des objectifs pédagogiques (Saraiva 2002). Elle permet la construction d'automates à l'aide d'expressions régulières.
Site : `www.di.uminho.pt/~jas/Research/HaLeX/HaLeX.html`.

Lintouch. Lintouch est un logiciel open source permettant de modéliser l'interface utilisateur d'un système de contrôle industriel. Dans le cadre fort général de ce logiciel, l'auteur fournit une petite bibliothèque d'automates à états finis écrite en C.
Site : `lintouch.org/download/lintouch-1.10/api-doc/libltfsm/index.html`.

REGAL. Bibliothèque écrite en C++, qui utilise massivement les Templates. Le code source est disponible, et les fonctionnalités de l'interface sont présentées dans (Bassino *et al.* 1999).
Site : `regal.univ-mlv.fr/`.

1.2 Machines classiques

AttiasFSTools. *Attia's Finite State Tools for Modern Standard Arabic*. Ensemble d'outils basés sur un encodage UTF-8 et disponibles exclusivement pour l'arabe. Ces outils dépendent d'un compilateur accessible aux détenteurs de (Beesley & Karttunen 2003).
Site : `www.attiaspace.com/getrec.asp?rec=fsttools`

Gdansk FSA/UTR. Deux bibliothèques, écrites en C++, dont le code source est disponible. L'une gère les automates (FSA) et l'autre, les transducteurs (UTR). Ces

[3] Haskell est un langage fonctionnel. Voir `www.haskell.org/`.

bibliothèques s'utilisent en ligne de commande. Les auteurs proposent également des dictionnaires pour l'allemand, le français et le polonais.
Site : www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/fsa.html.

OMAC FSM Library. Cette bibliothèque en C++ est fortement orientée objet. Elle n'est pas destinée aux traitements en langage naturel, mais est dédiée aux besoins des applications de contrôle qui peuvent être décrites comme une combinaison des modèles Moore et Mealy : il est donc question de traiter des machines qui associent des événements aux états (Moore) et aux transitions (Mealy). Ceci s'éloignant considérablement du domaine de cette thèse, nous renvoyons le lecteur à la littérature spécialisée (Mealy 1955, Moore 1956).
Site : www.isd.mel.nist.gov/projects/omacapi/Software/FiniteStateMachine/.

XFST. *Xerox Finite-State Tool.* Cette bibliothèque permet de manipuler des machines représentées au format textuel ou binaire. Elle intègre un compilateur d'expressions régulières. Elle est uniquement utilisable en ligne de commande et n'a pas été compilée pour plateforme embarquée.
Site : www.cis.upenn.edu/~cis639/docs/xfst.html.

1.3 Machines classiques et pondérées

Carmel. Bibliothèque implémentée en C++, selon le standard Boost [4]. La bibliothèque permet de charger des machines décrites dans un format textuel, mais ne propose pas de compilateur d'expressions régulières ni de règles de réécriture. La bibliothèque est utilisable en ligne de commande.
Site : www.isi.edu/licensed-sw/carmel/.

DFKIFSMToolkit. *DFKI Finite-State Machine Toolkit.* Cette bibliothèque, écrite en C++, présente tous les algorithmes nécessaires à la manipulation des machines à états finis, mais n'inclut pas de compilateur. Elle est décrite dans (Piskorski 2002), mais n'est pas téléchargeable.

FSM Library. Bibliothèque développée à AT&T (Mohri *et al.* 1997), qui peut être gratuitement téléchargée sous la forme d'un ensemble d'exécutables (un exécutable par fonction). Cette bibliothèque, très performante et très complète, est la première bibliothèque pondérée a avoir été développée. Elle n'est pas disponible pour plateforme embarquée.
Site : www.research.att.com/~fsmtools/fsm/.

GFSM. Bibliothèque assez complète écrite en C et dont le code source est disponible. L'auteur de cette bibliothèque récente signale que la dernière version (0.0.10) comporte encore de nombreux bugs. La compilation de la bibliothèque donne un ensemble d'exécutables utilisables en ligne de commande. En outre, les poids de cette bibliothèque

[4] Voir www.boost.org/.

sont de type float, ce qui limite la portabilité du code sur plateforme embarquée.
Site : `www.ling.uni-potsdam.de/~moocow/projects/gfsm/`.

Groningen FSA. Bibliothèque très complète, fournie en source ou sous la forme de binaires indépendants pour chaque opération définie. La bibliothèque est implémentée en Prolog et nécessite TCL/TK, ce qui limite sa portabilité.
Site : `www.let.rug.nl/~vannoord/Fsa/`.

MIT FST. Bibliothèque compilable principalement sous Linux. Aucune documentation n'est actuellement fournie, mais le code source est disponible. Développée en C++, cette bibliothèque utilise massivement les Templates, ce qui limite sa portabilité.
Site : `people.csail.mit.edu/ilh/fst/`.

openFst. Bibliothèque très récente présentée dans (Allauzen *et al.* 2007). L'objectif de cette bibliothèque est de donner un accès open source aux algorithmes de la FSM Library (cf. ci-dessus). Cette bibliothèque a été développée en C++ et utilise le standard STL (cf. ASTL ci-dessus), ce qui n'autorise pas sa compilation pour plateforme embarquée.
Site : `www.openfst.org`

PFST. Bibliothèque très complète présentée dans (Hanneforth 2006). Elle est fournie en open source avec un compilateur d'expressions régulières, de règles de réécriture et de grammaires hors-contexte. La bibliothèque, cependant, a été développée en C++ selon le standard STL (cf. ASTL ci-dessus). Elle n'est donc pas portable sur plateforme embarquée.
Site : `www.ling.uni-potsdam.de/~tom/fsm/`.

RWTH FSA. Bibliothèque qui présente tous les algorithmes nécessaires. Les évaluations réalisées par les auteurs montrent l'efficacité de l'implémentation (Kanthak & Ney 2004). La bibliothèque est en C++ et utilise les Templates pour rendre les classes génériques, ce qui limite sa portabilité. Elle est fournie sans compilateur.
Site : `www-i6.informatik.rwth-aachen.de/~kanthak/fsa.html`.

UCFSM. Bibliothèque écrite en C++ et accompagnée de quelques exécutables. Elle permet uniquement la manipulation d'*automates* pondérés. Aucune documentation n'est fournie.
Site : `www.linguistics.ucla.edu/people/grads/albro/software.html`.

Vaucanson. Bibliothèque écrite en C++, qui utilise le standard STL (cf. ASTL ci-dessus). Les algorithmes sont donc implémentés une seule fois et s'instancient selon le semi-anneau traité. Cette bibliothèque propose un format XML de description des machines. La présente des STL empêche la compilation pour plateforme embarquée.
Site : `www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Vaucanson`.

WFSC. *Weighted Finite State Compiler* développé à Xerox (Kempe *et al.* 2007). L'implémentation de la bibliothèque se divise en couches. La couche accessible au programmeur présente une interface proche d'un pseudocode, où le programmeur manipule des objets abstraits (ensembles, semi-anneaux, etc.). Les couches cachées implémentent,

en C++, les réalisations concrètes des objets abstraits. Cette bibliothèque a été développée autour de la bibliothèque XFST (cf. ci-dessus). Elle propose donc également de compiler des expressions régulières. Les expressions compilables peuvent être testées en ligne, mais la bibliothèque n'est pas téléchargeable.
Site : www.xrce.xerox.com/competencies/content-analysis/wfsc/home.en.html.

WFST. Extension de la bibliothèque ASTL (cf. ci-dessus), par ajout des transducteurs et des machines pondérées. La bibliothèque présente tous les algorithmes nécessaires. Comme dans ASTL, les algorithmes sont génériques, mais l'utilisation des STL empêche cette bibliothèque de tourner sur plateforme embarquée.
Site : membres.lycos.fr/adant/tfe/.

2 Les outils

Fadd. Basé sur Gdansk FSA (cf. ci-dessus), Fadd est une collection d'outils utiles au traitement automatique de la langue, comme la construction de modèles de langue ou d'analyseurs morphologiques (Daciuk & van Noord 2001).
Site : www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/fadd.html.

GRM. Construite autour de la FSM Library d'AT&T (cf. ci-dessus), cette collection d'outils propose un compilateur de règles de réécriture pondérées, un compilateur de grammaires hors-contexte, un compilateur de modèles de langue et un compilateur d'outils divers, comme des automates suffixes. Cette bibliothèque est décrite dans (Mohri 2001).
Site : www.research.att.com/~fsmtools/grm/

Hopskip. *The Johns Hopkins finite-state learning toolkit.* Cet environnement n'est pas encore disponible. L'objectif est de permettre l'entraînement de modèles probabilistes compilés sous la forme de machines à états finis. La base scientifique du projet est décrite dans (Eisner 2002a). Les applications visées sont, par exemple, l'analyse de texte, la reconnaissance de la parole ou l'extraction d'information.
Site : www.cs.jhu.edu/~jason/hopskip/.

INTEX. Il s'agit d'un environnement de développement linguistique développé à l'université de Franche-Comté (Silberztein v412). Cet environnement inclut des dictionnaires et des grammaires, et permet l'analyse de textes de plusieurs millions de mots en temps réel. Il inclut également les outils nécessaires à la création de ressources lexicales, morphologiques et syntaxiques. INTEX manipule uniquement des transducteurs classiques, les automates étant vus comme des transducteurs identitaires.
Site : msh.univ-fcomte.fr/intex/.

Jacaranda. Nouvel environnement écrit en Java, qui n'est pas encore téléchargeable, mais dont le principal intérêt devrait être la flexibilité : l'utilisateur pourra définir ses propres opérations et ses propres types. L'alphabet utilisé dans les machines pourra en outre

être défini par l'utilisateur.
Site : www.lme.die.supsi.ch/~pedrazz/jacaranda/index.html

LASH. L'objectif de cet environnement est de représenter et de manipuler des ensembles infinis et d'explorer des espaces d'états infinis, les ensembles infinis étant représentés à l'aide d'automates à états finis classiques. Au-delà de cet objectif principal, l'environnement permet également de construire et de manipuler des automates à états finis classiques.
Site : www.montefiore.ulg.ac.be/~boigelot/research/lash/.

LEXC. *The Finite-State Lexicon Compiler*. Il s'agit d'un compilateur de lexiques et de règles de Koskenniemi. Le résultat de la compilation est un transducteur lexical.
Site : www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/lexc-93/lexc93.html.

Lextools. Lextools est un compilateur permettant de créer des transducteurs pondérés à partir de descriptions linguistiques de haut niveau (strings, expressions régulières ou règles de réécriture). Le compilateur, qui accepte des fichiers de descriptions, a été développé autour de la FSM Library d'AT&T.
Site : www.research.att.com/~alb/lextools/.

MAP-3.1. Cet outil, écrit en Common Lisp, permet à l'utilisateur de modéliser, de tester et d'utiliser des lexiques et des analyseurs morphologiques. Sa documentation s'adresse tant aux non-programmeurs qu'aux programmeurs qui souhaitent intégrer ce système dans le leur. L'outil contient un analyseur morphologique et un dictionnaire anglais.
Site : www.cs.cmu.edu/~awb/pub/map/MAP3.1.tar.gz.

PC-KIMMO. Ce programme permet de produire et d'analyser des mots représentés selon le modèle de Koskenniemi. Un shell sert d'interface interactive entre l'utilisateur et les primitives d'une bibliothèque écrite en C. Le code source de la bibliothèque peut être téléchargé et intégré dans un programme créé par l'utilisateur. Dans cette bibliothèque, les règles de Koskenniemi et les lexiques sont implémentés sous la forme d'automates et de transducteurs.
Site : www.sil.org/pckimmo/.

REGI. Prononcé « *reggae* », cet outil ne crée pas de machines à états finis, mais permet l'interprétation d'expressions régulières en Prolog.
Site : www.ling.gu.se/~lager/regi.html.

SFST. *Stuttgart Finite State Transducer*. Environnement permettant la génération et la manipulation de transducteurs à états finis. L'environnement, présenté dans (Schmid 2005), comprend un langage de programmation (SFST-PL), une bibliothèque, un ensemble d'exécutables et un compilateur de règles de Koskenniemi.
Site : www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html.

Speech Tools Library. La *Edinburgh Speech Tools Library* est une collection d'outils, développés en C++, et d'exécutables permettant de construire et de manipuler des res-

sources linguistiques destinées au traitement de la parole. Parmi ces outils figurent deux exécutables, *wfst_build* et *wfst_run*, qui permettent respectivement la construction et l'utilisation de transducteurs pondérés. L'étape de construction est capable de gérer des expressions régulières, des grammaires régulières, des règles de réécriture et des règles de Koskenniemi.
Site : www.cstr.ed.ac.uk/projects/speech_tools/manual-1.2.0/x3787.htm.

SRILM. Il s'agit d'un ensemble formé de bibliothèques C++, de programmes et de scripts qui permettent la compilation et l'utilisation de modèles de langue (Stolcke 2002). Les graphes (de mots, de catégories) sont générés sous la forme de grammaires à états finis, qui peuvent être converties en machines à états finis dont le format respecte celui utilisé par la FSM Library d' AT&T (cf. ci-dessus).
Site : www.speech.sri.com/projects/srilm/.

Statechart. *The Boost Statechart Library.* Il s'agit d'un environnement permettant de convertir un graphe d'états au format UML [5] en un code C++ équivalent. Le code généré intègre des classes Templates prédéfinies dans l'environnement.
Site : www.boost.org/libs/statechart/doc/index.html.

Unitex. Environnement de développement linguistique développé à l'Institut Gaspard-Monge. Cet environnement inclut des grammaires et des dictionnaires similaires à ceux d'INTEX, et propose des fonctionnalités fort proches de celles d'INTEX. L'interface d'Unitex est écrite en Java, tandis que les algorithmes sont implémentés en C++.
Site : www-igm.univ-mlv.fr/~unitex/.

XFST2FSA. Cet outil permet de convertir des machines à états finis du format « XFST » au format « Groningen FSA » (cf. ci-dessus).
Site : www.cs.haifa.ac.il/~yaelc/research/xfst2fsa/.

[5] UML, *Unified Modeling Language*, est un langage graphique de modélisation des données et des traitements. Voir www.uml.org/.

Annexe B

Ovide : documentation

Des expressions régulières et des règles de réécriture aux machines à états finis

Richard Beaufort

version 1.4.2
(09 août 2007)

Développées dans le contexte de la phonologie générative, les règles de réécriture sont maintenant couramment employées dans de nombreux domaines du traitement du langage naturel (pré-processing, analyse morphologique, analyse syntaxique, conversion graphèmes-phonèmes...) et du traitement de la parole (synthèse de la parole par sélection d'unités non uniformes, reconnaissance de la parole).

En général, la description complète d'un domaine d'application nécessite un vaste ensemble de règles de réécriture. L'efficacité du système dépend de ce fait fortement de la méthode utilisée pour représenter ces règles. Johnson (1972) a démontré que les machines à états finis (FSMs), telles que les automates (FSAs) et les transducteurs (FSTs), permettent de modéliser les règles de réécriture. Mieux, ces machines constituent un outil idéal pour la modélisation des règles de réécriture : elles peuvent être représentées de manière compacte (Mohri 1994a), elles acceptent les opérations définies sur les langages réguliers, telles que l'union, la concaténation et l'étoile de Kleene (Roche & Schabes 1997), et peuvent être combinées par composition (Pereira & Riley 1997, Mohri *et al.* 1996), un type particulier d'intersection qui facilite le processus de compilation des règles de réécriture.

Ovide, qui est basé sur une bibliothèque originale de machines à états finis développée à Multitel ASBL, permet de compiler des dictionnaires, des langages réguliers et des règles de

réécriture en machines à états finis équivalentes. Pour répondre aux besoins des applications réelles, Ovide permet la définition de règles de réécriture pondérées et/ou optionnelles.

1 A propos des expressions régulières

1.1 Quelques définitions

Définition 1.1 (Alphabet)**.** *Un alphabet est un ensemble de symboles.*

Par exemple, $\{a, b, c, \ldots, y, z\}$ est l'alphabet des lettres minuscules, tandis que $\{0, 1, 2, \ldots, 8, 9\}$ est l'alphabet des chiffres.

Définition 1.2 (Langage régulier)**.** *Un langage régulier est un langage formel qui est clos sous la concaténation, l'intersection, la complémentation, l'union et l'étoile de Kleene.*

Notons qu'une classe de langages est dite close sous une opération donnée lorsque l'application de cette opération, à un langage qui appartient à la classe, donne un langage appartenant également à cette classe. Par exemple, le résultat de la concaténation de deux langages réguliers est un langage régulier.

Définition 1.3 (Expression régulière)**.** *Une expression régulière est une description syntaxique d'un langage régulier.*

Cette description syntaxique définit un langage en compréhension, où les symboles d'un alphabet Σ donné sont combinés à l'aide d'opérateurs prédéfinis.

Initialement, chaque opérateur correspond à une opération régulière sous laquelle les langages réguliers sont clos : concaténation, intersection, complémentation, union et étoile de Kleene. Cependant, certaines combinaisons de ces opérations peuvent être facilement exprimées à l'aide d'un seul opérateur. De ce fait, la liste des opérateurs définis sur les expressions régulières est un peu plus longue que la liste des opérations régulières.

Définition 1.4 (String vide)**.** *La string vide est la string ne comportant aucun symbole. On l'appelle* epsilon *et on la note* ϵ*.*

Définition 1.5 (Monoïde libre)**.** *Etant donné un alphabet* Σ*, le monoïde libre* Σ^* *est l'ensemble formé de la string vide et de toutes les strings construites à partir de* Σ*.*

Donc, le monoïde libre *est* un langage régulier, et tout langage régulier défini sur Σ est égal ou inclus dans Σ^*.

1.2 Opérateurs définis dans Ovide

Notons que certains opérateurs ne s'appliquent qu'à des symboles ou des strings (symboles combinés uniquement par concaténation). Dans la liste ci-dessous, la notation suivante est utilisée :

- `a`, `b`, `c`, : les lettres minuscules représentent des symboles.
- `X`, `Y` : les lettres majuscules représentent des strings.
- `REG` : représente une expression régulière.
- `OP` : représente un simple opérateur.

`REG*`	:	0, 1 ou plusieurs `REG`.
`REG+`	:	au moins un `REG`.
`REG?`	:	0 ou un `REG`.
`REG{n,m}`	:	au moins n, maximum m `REG`.
`REG{n}`	:	n `REG`.
`REG{n,}`	:	n `REG` ou plus.
`REG{,m}`	:	de 0 à m `REG`.
`(REG)`	:	regroupement (modification de la précédence standard des opérateurs, cf. ci-dessous).
$\texttt{REG}_1 \cdot \texttt{REG}_2$	:	*concaténation* de $\texttt{REG}_1$ et $\texttt{REG}_2$: $\texttt{REG}_1\texttt{REG}_2$. Notons que cet opérateur n'est jamais écrit. Les expressions sont directement côte-à-côte.
$\texttt{REG}_1 \mid \texttt{REG}_2$	:	*union* de $\texttt{REG}_1$ et $\texttt{REG}_2$. Attention à la précédence des opérateurs : `ab|cd` = `(ab)|(cd)` et non `a(b|c)d`.
`[abc]`	:	égal à `(a|b|c)`.
`[a-z]`	:	égal à `(a|b|...|y|z)`.
`[^abc]`	:	pas `abc` (tous les symboles sauf `abc`).
`^REG`	:	`REG` doit correspondre au début de la phrase.
`REG$`	:	`REG` doit correspondre à la fin de la phrase.
`"X"`	:	expression brute, pas d'interpretation de `X`, qui est une string qui peut contenir des opérateurs, vus comme de simples symboles.
`\OP`	:	précède un opérateur `OP` qui doit être interprété comme un symbole.

1.3 Précédence des opérateurs dans Ovide

(1) `"X"` `\OP`
(2) `^REG` `REG$`
(3) `(REG)`
(4) `[abc]` `[^abc]`
(5) `REG*` `REG+` `REG?` `REG{n,m}` `REG{n}` `REG{n,}` `REG{,m}`
(6) $\texttt{REG}_1 \cdot \texttt{REG}_2$ $\texttt{REG}_1 \mid \texttt{REG}_2$

Note. Les opérateurs d'une même ligne ont la même précédence.

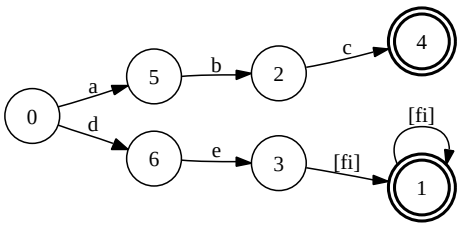
1.4 Strings prédéfinies dans Ovide

1. *N'importe quel alphabet* :
   - `.` : n'importe quel symbole
   - `\NUM` : où `NUM` est un nombre, l'index d'un symbole de l'alphabet
   - `\x[0-F][0-F]` : notation hexadécimale d'un symbole
2. *Seulement en ASCII* :
   - `\d` : n'importe quel chiffre
   - `\w` : n'importe quel caractère de mot
   - `\s` : n'importe quel blanc (espace, tabulation)
   - `\D` : tout sauf un chiffre
   - `\W` : tout sauf un caractère de mot
   - `\S` : tout sauf un blanc
   - `\a` : sonnette, alerte
   - `\b` : *backspace*
   - `\f` : *form feed*
   - `\n` : nouvelle ligne
   - `\r` : retour chariot
   - `\t` : tabulation horizontale
   - `\v` : tabulation verticale

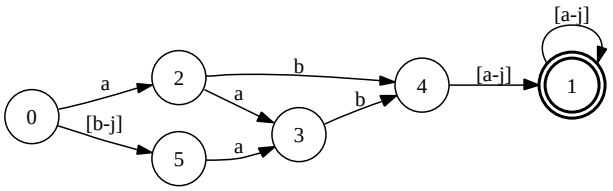
1.5 Exemples

Voici quelques exemples d'expressions régulières, leur « signification » et leur représentation sous la forme d'une machine à états finis. Tous les exemples utilisent le même alphabet : `[a-j]`.

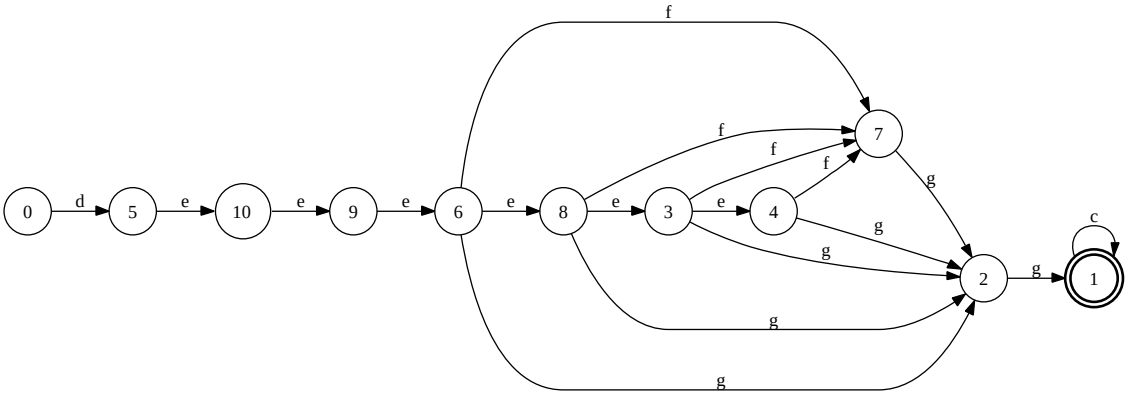
- `abc|de(f|i)+`
  abc **ou** *de* suivis par au moins un *f* ou *i*.



- `.?ab.+`
  0 ou 1 symbole de l'alphabet, suivi par *ab*, suivis par au moins un symbole de l'alphabet.



- `de{3,6}f?g{2}c*`
  d, suivi par 3 à 6 *e*, suivis par 0 ou 1 *f*, suivi par 2 *g*, suivis par 0, 1 ou plusieurs *c*.



2 A propos des règles de réécriture

2.1 Règles classiques

De manière générale, une règle de réécriture décrit une transduction entre deux expressions régulières. Cependant, les règles de réécriture prennent généralement la forme suivante :

$$\phi \rightarrow \psi : : \lambda \_ \rho \tag{2.1.1}$$

qui signifie que ϕ doit être réécrit ψ quand il est entouré par λ et ρ. Dans cette notation, λ, ψ, ϕ et ρ sont des expressions régulières. La transduction dont nous parlions précédemment est réalisée entre la partie haute de la règle, $\lambda\ \phi\ \rho$, et la partie basse de la règle, $\lambda\ \psi\ \rho$.

Notons que deux contraintes sont exprimées sur des ensembles de règles :

1. Les règles sont ordonnées de la plus spécifique à la plus générale, de sorte qu'une règle donnée ne puisse s'appliquer que si aucune règle, plus spécifique et plus appropriée, n'a été rencontrée précédemment.

2. Tout contexte d'une règle (λ, ρ) qui est la cible d'une autre règle sera également réécrit : en somme, toutes les règles appropriées pour une entrée donnée lui seront appliquées, simultanément ou récursivement.

Ovide respecte ces contraintes.

2.2 Règles pondérées

De nombreuses applications ont besoin d'une solution permettant de classer toutes les solutions possibles pour une entrée donnée. Ceci est particulièrement vrai pour les applications qui impliquent un traitement automatique de la langue, comme la correction orthographique, l'analyse syntaxique ou la reconnaissance de la parole. Mohri & Sproat (1996) furent les premiers à proposer une extension pondérée des règles de réécriture. Cette extension est également disponible dans Ovide.

Les règles de réécriture pondérées prennent la forme suivante :

$$\phi \rightarrow \psi : : \lambda \_ \rho / \omega \tag{2.2.1}$$

qui signifie que le remplacement $\phi \rightarrow \psi$, réalisé quand ϕ est entouré par $\lambda \_ \rho$, se voit attribuer le poids ω.

2.3 Règles optionnelles

Dans certaines applications, comme la correction orthographique, on désire **aligner des strings** de symboles. L'alignement de séquences est classiquement résolu par *Dynamic Time Warping* (Wagner & Fisher 1974, Myers & Rabiner 1981). Pourtant, les machines à états finis constituent une alternative intéressante. L'idée est de créer un transducteur à états finis F qui autorise des **transformations optionnelles**. Comme le note Mohri (2003), F peut être vu comme un **filtre** entre deux strings s_1 et s_2 à aligner :

$$s_1 \circ F \circ s_2 \tag{2.3.1}$$

Pour ce type d'applications, des règles pondérées optionnelles sont nécessaires. Elles prennent la forme suivante :

$$\phi \ ?\rightarrow \ \psi : : \lambda \_ \rho / \omega \tag{2.3.2}$$

qui signifie que le remplacement $\phi \ ?\rightarrow \ \psi$ est facultatif.

2.4 La string vide

Les alphabets utilisés par Ovide sont compilés par la bibliothèque FSM de Multitel ASBL, où la string vide vaut toujours 0 et où l'index du premier symbole d'un alphabet est toujours 1.

Ceci étant posé, Ovide peut exprimer la string vide de deux façons :

- notation décimale : `\0`
- notation hexadécimale : `\x00`

Voici quelques exemples de règles d'*insertion* et de *suppression*, à l'aide de la notation hexadécimale :

- Insertion : `\x00 → a : : b _ c / 0.1`
- Suppression : `a → \x00 : : b _ c / 0.1`

3 Les sections d'un fichier

La Table B.1 présente la liste des sections qui peuvent constituer un fichier Ovide. Pour respecter la syntaxe d'Ovide, les noms de sections doivent être indiqués entre crochets droits. Dans un fichier Ovide, une ligne consacrée à une section ne peut contenir aucune autre information.

Type de section	**Mode 1**	**Mode 2**	**Nom**
Informations générales	!	!	`[INFO]`
Classes d'entrée	?		`[CLASSIN]`
Classes de sortie	?		`[CLASSOUT]`
Langage d'entrée	*		`[LANGIN]` ou `[DICIN]`
Règles de réécriture	*		`[RULE]` ou `[COND]`
Langage de sortie	?		`[LANGOUT]` ou `[DICOUT]`
Inclusion	?	!	`[INCLUDE]`
Compilation modifiée		?	`[COMPILE]`

TAB. B.1: Sections d'Ovide

Dans la table, des symboles différencient l'importance des sections :

- ! : la section est obligatoire.
- ? : la section est facultative.
- * : ce symbole figure en face de plusieurs sections facultatives, dont au moins une doit être présente.
- Lorsque rien n'est indiqué, la section est interdite.

Certaines sections (les langages et les règles) peuvent prendre deux valeurs mutuellement exclusives. Par exemple, le langage d'entrée est soit un `LANGIN`, soit un `DICIN` (cf. ci-dessous), mais pas les deux en même temps.

La seule section toujours obligatoire est celle des informations générales. Les autres sections, globalement, sont combinées selon deux modes possibles :

1. Le mode 1 est le schéma classique. Son objectif est principalement de compiler un langage et/ou des règles de réécriture. Des règles peuvent être compilées sans qu'un langage soit précisé. Dans ce cas, le langage par défaut est le monoïde libre Σ^*. Si un langage accompagne des règles, les règles seront appliquées à ce langage. Dans ce schéma classique, le mode de compilation par défaut est la *composition*. Les langages, les règles et les fichiers inclus sont composés dans l'ordre suivant :

$$\begin{gathered}(\text{LANGIN}|\text{DICIN}) \\ \circ \\ (\text{RULE}|\text{COND}) \\ \circ \\ (\text{LANGOUT}|\text{DICOUT}) \\ \circ \\ \text{INCLUDE}_1 \\ \circ \\ \dots \\ \circ \\ \text{INCLUDE}_n\end{gathered}$$

2. Le mode 2 travaille sur des FSMs correspondant à des fichiers inclus. L'objectif est ici de définir une compilation particulière, qui combine les FSMs inclus autrement que dans une simple composition. Le schéma peut cependant être employé sans section de compilation. Dans ce cas, les FSMs inclus seront simplement composés dans l'ordre d'inclusion.

Enfin, on constate qu'il est possible d'utiliser deux alphabets et de décrire des classes et des langages sur ces deux alphabets. Il n'est cependant pas obligatoire de travailler sur deux alphabets. Par exemple, des règles de réécriture peuvent transformer des symboles ASCII en d'autres symboles ASCII. Dans ce cas, on emploie un seul alphabet, et seules les sections relatives à l'alphabet d'entrée sont autorisées.

3.1 Informations générales

La section `INFO` accepte les champs suivants :

- `ALPHAIN` et `ALPHAOUT`
- `SUBALPHAIN` et `SUBALPHAOUT`
- `APPLY`
- `FILTER`

3.1.1 ALPHAIN et ALPHAOUT

`ALPHAIN` est **nécessaire** tandis qu'`ALPHAOUT` est **facultatif**. Par défaut, `ALPHAOUT` est égal à `ALPHAIN`.

```
ALPHAIN  =  ASCII
            ALPHA
            NUM
            ALPHANUM
            alphabet.symbol
```

1. `ASCII` : caractères ASCII (256 symboles, ISO-8859-1)
2. `ALPHA` : caractères alphabétiques, c'est-à-dire [a-z], [A-Z] et leurs correspondants accentués (115 symboles)
3. `NUM` : de 0 à 9 (10 symboles)
4. `ALPHANUM` : les symboles `ALPHA` et `NUM` ensemble (125 symboles)
5. `alphabet.symbol` : un alphabet défini par l'utilisateur, un symbole par ligne. La première ligne peut définir la string vide ϵ :

   `\EPSILON=`$\cdots$

   où $\cdots$ est une string de maximum 8 caractères. Par défaut, la string vide est \x00, qui n'est pas si difficile à utiliser...

Attention. Ovide est ***case sensitive*** et doit souvent comparer plusieurs alphabets (au niveau de leurs noms et de leurs symboles). De ce fait, pour un même alphabet, utilisez toujours la même casse dans vos déclarations.

3.1.2 SUBALPHAIN et SUBALPHAOUT

Souvent, plusieurs FSMs travaillent sur le même alphabet, mais n'en utilisent pas la même partie. Dans ce cas, utiliser l'alphabet complet lors de la compilation séparée de chaque FSM ralentit le processus et n'améliore pas le résultat final.

La solution *pourrait être* de définir un alphabet particulier pour chaque FSM, ne contenant que les symboles réellement nécessaires à la compilation. Cependant, cette solution ne fonctionne pas, parce que les différents FSMs compilés ne pourront être composés ensemble : les symboles communs des différents (sous-)alphabets définis ne partageront pas la même valeur numérique en interne. Par exemple, le symbole "A", qui a le valeur numérique 65 dans l'alphabet ASCII, aura la valeur 1 dans un sous-alphabet dont il sera le premier symbole, mais la valeur 32 dans un sous-alphabet dont il sera le 32^e symbole...

`SUBALPHAIN` et `SUBALPHAOUT` sont très utiles dans ce cas. Ils permettent à l'utilisateur de définir *en extension* l'ensemble des symboles de `ALPHAIN` et `ALPHAOUT` qui sont véritablement utiles, tout en conservant les valeurs numériques de l'alphabet complet.

L'utilisation de `SUBALPHAIN` et `SUBALPHAOUT` est très simple : il suffit de donner les symboles utiles entre crochets droits.

Par exemple, si l'alphabet d'entrée est l'`ASCII` (256 symboles), mais que vous n'employez que les symboles allant de `A` à `Z`, de `a` à `n` et l'espace, votre sous-alphabet peut être défini comme suit :

```
ALPHAIN = ASCII
SUBALPHAIN = [A-Za-n\ ]
```

où "\ " est l'espace.

La définition d'un sous-alphabet de sortie n'est autorisée que si l'alphabet de sortie diffère de l'alphabet d'entrée. Par exemple, sur un alphabet de sortie fait de phonèmes, vous pourriez définir le sous-alphabet suivant :

```
ALPHAOUT = Phonemes.symbol
SUBALPHAOUT = [ieEa a~ e~ 9~ o~ pbm]
```

Attention. La définition d'un sous-alphabet n'est autorisée qu'après que l'alphabet correspondant ait été déclaré. L'exemple suivant vous montre une mauvaise utilisation de la définition d'un `SUBALPHAIN` :

```
SUBALPHAIN = [A-Za-n\ ]
ALPHAIN = ASCII
```

3.1.3 APPLY

`APPLY` est un champ **facultatif**.

```
APPLY  =  OUTPUT
          INPUT
```

1. `OUTPUT` : valeur par défaut, qui correspond au comportement standard de la composition. Etant donné deux FSMs T_1 et T_2 composés ensemble $(T_1 \circ T_2)$, la composition standard ne tient compte que de l'interface commune (*i.e.* l'alphabet de *sortie* de T_1).
2. `INPUT` : l'idée, dans ce cas, est qu'une règle ne peut être appliquée ***que si*** le symbole d'entrée n'a pas été précédemment modifié par une autre règle. La composition $(T_1 \circ T_2)$ tient dès lors également compte de l'alphabet d'*entrée* de T_1.

Exemple. Nous avons 2 règles :

```
rule 1  =  a -> b
rule 2  =  b -> c
```

Nous avons un état avec 3 transitions :

```
a :a
b :b
a :b
```

Comportement avec `APPLY=OUTPUT` :

```
a :a  →  a :c
b :b  →  b :c
a :b  →  a :c
```

Comportement avec `APPLY=INPUT` :

```
a :a  →  a :b
b :b  →  b :c
a :b  =  a :b
```

3.1.4 FILTER

`FILTER` est un champ **facultatif**.

```
FILTER = filtre.fsm
```

Pendant le processus de compilation d'une règle, les symboles de ϕ et de ψ sont automatiquement alignés, par calcul de leur produit cartésien ($\phi \times \psi$). Par exemple,

```
couvent -> k u~ v a~
```

sera automatiquement aligné :

```
c  o   u  v   e     n     t
k  u~  v  a~  \x00  \x00  \x00
```

Le filtre facultatif permet de préciser au compilateur une liste d'**alignements préférentiels**. Par exemple, v est mieux aligné avec v qu'avec u...

Le filtre est un FSM. Il peut donc être compilé par Ovide à partir d'un fichier contenant une simple liste de règles de réécriture pondérées et **facultatives**. Pour un exemple complet d'un fichier Ovide utilisant un filtre, consultez la Section 6.3. Sur notre exemple précédent, le meilleur alignement possible est :

```
c  o   u     v  e   n     t
k  u~  \x00  v  a~  \x00  \x00
```

Attention. Le FSM résultant **n'est pas toujours** plus petit parce qu'un filtre a été utilisé. Par contre, le résultat est plus *logique*...

3.2 Les classes d'entrée

La section `CLASSIN` permet la déclaration de classes et de marqueurs qui seront utilisés plus loin dans le fichier. Une classe est une expression régulière, tandis qu'un marqueur est interprété comme un symbole particulier.

3.2.1 Classes standard

Une classe est déclarée sur une ligne. La première string de la ligne est le nom de la classe, séparé par des blancs (espaces ou tabulations) de la définition de la classe. En voici quelques exemples :

```
VOWEL       a|e|i|o|u|y
CONS        b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|z
```

Pour appeler une classe, son nom est mis entre crochets. Par exemple,

```
<VOWEL>
```

Une classe peut être utilisée aussitôt après sa déclaration. Donc, par exemple, dans la déclaration d'une autre classe :

```
VOWEL       a|e|i|o|u|y
CONS        b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|z
CHAR        [<CONS><VOWEL>]
```

le nom de la classe est une string combinant uniquement des caractères ASCII alphanumériques : [a-z], [A-Z], [0-9] et "-". Notons que "_" et les caractères accentués sont **interdits**.

```
EPS
NAME
VERBO1
TASK-1
```

La définition de la classe est une expression régulière classique, faite de symboles et d'opérateurs :

```
abc
ab|c
(a?b)|c
[a-z]+
```

Bien sûr, cette section n'accepte que des symboles appartenant à l'alphabet d'entrée...

3.2.2 Cas particulier : les marqueurs

Un marqueur est appelé *comme une classe*, entre crochets. C'est la raison pour laquelle nous permettons de définir les marqueurs dans la section des classes. Par exemple,

```
<MARKER>
```

Définition 3.1 (Marqueur)**.** *Un* marqueur *est un symbole, extérieur à l'alphabet utilisé, que l'on insère dans une règle de réécriture afin d'identifier un phénomène et d'en suivre l'évolution.*

Trois types de marqueurs sont définis : le *déclencheur*, le *masqueur* et le *bloqueur*.

Définition 3.2 (Déclencheur)**.** *Le* déclencheur *est un marqueur qui permet de signaler de manière non ambiguë qu'une condition d'application a été rencontrée, de manière à déclencher, en toute sécurité, l'application d'une ou de plusieurs règles de réécriture.*

Le déclencheur δ est donc inséré par une règle, qui constate qu'une condition est remplie. D'autres règles peuvent ensuite préciser δ, afin de conditionner leur application à sa présence. Dans l'exemple suivant,

(1) $\epsilon \rightarrow \delta : :[condition]$
(2) $a \rightarrow b : :\_\ \delta\, def$
(3) $f\,\delta \rightarrow g : :\_\ def$
(4) $\delta \rightarrow \epsilon$

la première règle insère le déclencheur. La seconde s'applique en présence du déclencheur, mais ne le réécrit pas. Ceci signifie qu'une autre règle, qui préciserait le même déclencheur, pourrait également s'appliquer. La troisième règle, par contre, réécrit le déclencheur, ce qui empêche l'application d'autres règles par la suite. La quatrième règle supprime le déclencheur éventuellement encore présent, et devenu inutile.

Définition 3.3 (Masqueur)**.** *Le* masqueur *est un marqueur qui prend temporairement la place d'une expression régulière, afin d'éviter qu'une règle de réécriture ne s'applique à tort sur cette expression.*

Typiquement, un masqueur μ est utilisé à la place de la réécriture (ψ) de la règle. En somme, la cible de la règle est réécrite par un masqueur μ attribué à une réécriture

particulière. Il est donc nécessaire qu'une autre règle, située à la fin de l'ensemble de règles, convertisse μ en la réécriture qu'il masque. En voici un exemple :

(1) $a \rightarrow \mu : :c\_d$
(2) $b \rightarrow e : :c\_d$
(3) $\mu \rightarrow b$

Dans l'exemple, la première règle masque la réécriture b à l'aide du masqueur μ. Ceci évite à la deuxième règle, dont la cible est b et qui concerne le même contexte que la première règle, de pouvoir s'appliquer sur une entrée qui aurait été réécrite par la première règle. La troisième règle peut enfin, en toute sécurité, convertir μ en b, sans risque de réécriture erronée.

Définition 3.4 (Bloqueur)**.** *Le* bloqueur *est un marqueur qui s'insère temporairement entre deux expressions régulières, afin d'éviter la formation d'une nouvelle expression régulière sur laquelle une règle de réécriture pourrait s'appliquer à tort.*

Le bloqueur est par exemple utile dans le cas d'une règle qui supprime une cible, étant donné que cette suppression peut entraîner la formation d'un contexte favorable à l'application inappropriée d'une autre règle. Comme le masqueur, le bloqueur β doit être réécrit, lorsque tout risque de réécriture abusive a disparu :

(1) $a \rightarrow \beta : :c\_d$
(2) $c \rightarrow e : :\_d$
(3) $\beta \rightarrow \epsilon$

Mise en œuvre dans Ovide. La manière dont un marqueur est déclaré ne dépend pas de son type. Le type n'est jamais spécifié, mais se découvre par la manière dont le marqueur est utilisé.

La première string de la ligne est le nom du marqueur, séparé par des blancs (espaces ou tabulations) de la définition du marqueur. La définition d'un marqueur *n'est pas* une expression régulière, mais un simple *identifiant* précédé de "&" :

```
ATRIG    &1
AMASK    &2
AFREZ    &3
```

3.3 Les classes de sortie

La section `CLASSOUT` permet la définition de classes sur l'alphabet de sortie. Ces classes seront utilisées plus loin dans le fichier Ovide.

3.4 Les langages d'entrée

La section `LANGIN` contient des expressions régulières (une par ligne) définies sur l'alphabet d'entrée. S'il y a plusieurs expressions régulières, le langage régulier correspondra à l'union des différentes expressions.

La section `DICIN` contient des strings (une par ligne) définies sur l'alphabet d'entrée et qui ne contiennent **aucun** opérateur. En somme, chaque string est un **simple mot**. Le résultat est donc un dictionnaire correspondant à l'union de ces mots.

Bon à savoir. Le dictionnaire est intéressant, parce que très rapide à compiler.

3.5 Les règles de réécriture

3.5.1 RULE

Voici les différentes formes que peuvent prendre les règles de réécriture dans Ovide :

(1) `A -> B :: C _ D / W`

(2) `A -> B :: C _ D`

(3) `A :: C _ D / W`

(4) `A -> B / W`

(5) `A -> B`

(6) `A / W`

(7) `A?-> B :: C _ D / W`
`A?-> B :: C _ D`
`A?-> B / W`
`A?-> B`

Ces règles s'interprètent comme suit :

(1) `A` est réécrit `B` et reçoit le poids `W` quand entouré de `C` et `D`
`A`, `C` et `D` : expressions régulières définies sur des symboles d'**entrée**
`B` : expression régulière définie sur des symboles de **sortie**
`W` : un poids

(2) `A` est réécrit `B` quand entouré de `C` et `D`. Pas de poids.

(3) `A` reçoit le poids `W` quand entouré de `C` et `D`. Pas de réécriture.

(4) `A` est toujours réécrit `B` et reçoit le poids `W`.

(5) `A` est toujours réécrit `B`. Pas de poids.

(6) `A` reçoit le poids `W`. Pas de réécriture. Permet de pondérer un langage.

(7) `?->` signifie *règle facultative.*
De ce fait, le résultat est que 2 chemins sont créés pour `A` dans la machine.
Le premier ne change pas `A`, tandis que le second applique la règle.

Notes à propos des opérateurs de début (`^`) et de fin (`$`) :

- La partie `C` (contexte gauche) est la seule à pouvoir présenter `^` : ce symbole exprime que le contexte gauche *doit* commencer la string. De ce fait, `^` doit être *au début* de `C`.
- La partie `D` (contexte droit) est la seule à pouvoir présenter `$` : ce symbole exprime que le contexte droit *doit* terminer la string. De ce fait, `$` doit être *à la fin* de `D`.
- Ces symboles ne peuvent apparaître dans aucune autre partie. Rappelez-vous cependant qu'entre crochets droits, `[^...]` signifie « tout sauf ... », et peut apparaître dans n'importe quelle partie.

Note à propos des poids :

Les poids de nos règles sont des probabilités ou des distances. Attention, cependant : la bibliothèque de FSMs utilisée est définie sur le semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$. Donc, dans la bibliothèque de FSMs,

1) Un petit poids est meilleur qu'un grand poids, et 0 est le meilleur poids.
2) Un poids négatif est **interdit. Le résultat d'un algorithme de *recherche du meilleur chemin* sera incorrect si des poids négatifs sont définis.**
3) Les poids sont **additionnés** le long d'un chemin donné.

De ce fait, les probabilités doivent avoir été converties sous la forme de logarithmes négatifs :

$$-\log(P(w)) \tag{3.5.1}$$

3.5.2 COND

Section identique à `RULE`, mais qui indique que **toutes** les règles sont facultatives. L'opérateur de réécriture n'est pas important ici. Il peut s'agir aussi bien de `?->` que de `->`.

3.6 Les langages de sortie

La section `LANGOUT` est identique à `LANGIN`, mais définie sur l'alphabet de sortie.

La section `DICOUT` est identique à `DICIN`, mais définie sur l'alphabet de sortie.

3.7 L'inclusion

3.7.1 Principe

La section `INCLUDE` permet à l'utilisateur d'inclure d'autres fichiers (fichiers Ovide ou FSMs pré-compilés). Ces fichiers seront **composés** avec l'ensemble de règles (ou le langage) du fichier courant. Bien sûr, le mécanisme de la composition implique **certaines contraintes** concernant les alphabets utilisés par les différents fichiers : tous doivent avoir une **interface compatible** pour la composition.

Quel principe respecter ? Un fichier F_1, travaillant sur la paire d'alphabets {`A:B`}, inclut un autre fichier F_2. Ce fichier F_2 travaille sur une autre paire d'alphabets. Dans ce cas, l'**alphabet d'entrée** de F_2 **doit être** `B`, tandis que son alphabet de sortie importe peu (`B`, `A` ou un nouvel alphabet `C`). On notera la paire d'alphabets de F_2 {`B:out`}.

Quel est le résultat ? $F_1 \circ F_2$ travaillera sur la paire d'alphabets {`A:out`} :

```
{A:B} ∘ {B:A}  →  {A:A}
{A:B} ∘ {B:B}  →  {A:B}
{A:B} ∘ {B:C}  →  {A:C}
```

Notes :

1. Bien sûr, **un fichier peut inclure plus d'un fichier**. Dans ce cas, rappelez-vous que la paire d'alphabets de la machine en construction *change* à chaque nouvelle inclusion.
2. **Un fichier inclus peut lui-même inclure des fichiers**. En soi, le mécanisme d'inclusion n'est pas borné. Il permet de modéliser des machines correspondant à des langages multi-niveaux. Bien sûr, à la fin, la machine compilée ne travaille que sur 2 alphabets, l'alphabet d'entrée du premier fichier et l'alphabet de sortie du dernier fichier inclus.
3. Les champs `ALPHAIN` et `ALPHAOUT` d'un fichier ne doivent pas être influencés par le fait que ce fichier en inclut un ou plusieurs autres. Il faut juste que le fichier ait une **interface compatible** avec le premier fichier inclus. Le champ `ALPHAOUT` ne doit donc être spécifié que si **le fichier lui-même** emploie un alphabet de sortie.
4. Il arrive par contre qu'un fichier ne définisse ni langage, ni règles, mais présente une section `INCLUDE`. Dans ce cas, le fichier en question doit préciser l'alphabet d'entrée du premier fichier inclus.

Un exemple correct. F_1, travaillant sur {`A:B`}, inclut :
F_2{`B:C`}
F_3{`C:D`}
F_4{`D:A`}

Dans ce cas,
- F_2 travaille sur {B:C}. La paire d'alphabets devient {A:C}.
- F_3 travaille sur {C:D}. La paire d'alphabets devient {A:D}.
- F_4 travaille sur {D:A}. La paire d'alphabets devient {A:A}.

Un exemple incorrect. F_1, travaillant sur {A:B}, inclut :

F_2{B:C}
F_3{B:C}
F_4{C:D}

Dans ce cas,
- F_2 travaille sur {B:C}. La paire d'alphabets devient {A:C}.
- F_3 travaille sur {B:C}, **comme** F_3. Ceci va **interrompre** la compilation, parce que {B:C} **n'est pas compatible** avec la paire d'alphabets courante, {A:C}.

Comment résoudre ce problème ? En fait, F_2 et F_3 doivent constituer un seul fichier...

3.7.2 Syntaxe

Le principe général est que les fichiers inclus sont précisés ***sans extension***. Par exemple,

```
categories
syntax
```

Le système essaie d'abord d'ouvrir un FSM pré-compilé (ici, `"categories.fsm"` et `"syntax.fsm"`). Si le FSM n'existe pas, le système essaie d'ouvrir un fichier Ovide, dont l'extension **doit être** `".regexp"` (donc, `"categories.regexp"` et `"syntax.regexp"`). La compilation sera naturellement interrompue si aucun des fichiers n'existe.

Note. Le fichier Ovide est toujours automatiquement recompilé si l'option `"-r"` (pour *rebuild*) n'est pas spécifiée (cf. Section 5.2).

Si vous désirez charger un FSM pré-compilé qui n'a pas de fichier Ovide correspondant, spécifiez l'extension `".fsm"`, comme ceci :

```
categories.fsm
syntax.fsm
```

Attention. Dans ce cas, Ovide s'attend à ce que le FSM soit un binaire qui connaît ses **propres alphabets**. Pour savoir comment ajouter des alphabets dans un FSM, consultez la documentation de la bibliothèque.

3.8 La compilation particulière

Le processus de compilation par défaut est une composition (∘) de 3 machines (si elles existent, bien sûr) :

```
(LANGIN|DICIN) ∘ (RULE|COND) ∘ (LANGOUT|DICOUT)
```

La section `COMPILE` permet à l'utilisateur de modifier ce comportement par défaut, en précisant la compilation désirée.

Le principe est d'utiliser des ***fichiers pré-compilés***, définis dans la section `INCLUDE`. Par exemple,

```
categories_A  # works on {A:B}
categories_B  # works on {A:B}
syntax        # works on {B:C}
```

Pour utiliser ces fichiers dans la section `COMPILE`, il suffit de préfixer leurs noms par `@` :

```
@categories_A
```

La syntaxe de cette section est équivalente à celle des expressions régulières. Un opérateur supplémentaire est disponible, cependant, pour permettre à l'utilisateur de réaliser une **composition** : ˚.

Notes :

1. Dans notre exemple, `categories_A` **ne peut être composé** avec `categories_B`, parce que les deux fichiers travaillent sur la **même paire d'alphabets**. Par contre, ils peuvent être combinés par union ou par concatenation, par exemple.
2. *A contrario*, `categories_A` **doit être composé** avec `syntax`. Les deux FSMs **ne peuvent être combinés** par union ou par concatenation, parce qu'ils travaillent sur des paires d'alphabets différentes.

Voici un exemple de règle de compilation correcte manipulant ces 3 fichiers :

```
(@categories_A|@categories_B) ˚ @syntax
```

Vous pouvez également déclarer plusieurs lignes de compilation. Dans ce cas, toutes les lignes seront combinées par union. Par exemple,

```
@categories_A ˚ @syntax
@categories_B ˚ @syntax
```

sont combinées par union. Le résultat obtenu est équivalent à celui de l'exemple précédent.

Attention. Les seules section permises avec une section `COMPILE` sont les sections `INFO` et `INCLUDE`.

4 Quelques opérateurs particuliers

4.1 L'opérateur *strict*

Certaines règles sont difficiles à décrire de manière synthétique. En voici un exemple :

```
i a → a i
i b → b i
i c → c i
...
```

Dans ce cas, quel que soit le symbole qui suit i, i et le symbole sont inversés. La règle suivante, plus synthétique, ne convient pas :

```
i . → . i
```

parce qu'elle signifie « i suivi de n'importe quel symbole » devient « **n'importe quel symbole** suivi de i » :

```
i a → . i
i b → . i
i c → . i
...
```

Pour résoudre ce problème, nous avons défini l'**opérateur *strict*** : `%`. Cet opérateur, lorsqu'il précède un **ensemble de symboles**, comme `.` ou une *classe* (cf. Sections 3.2 et 3.3), signifie que chaque symbole de l'ensemble doit être traité séparément. Grâce à cet opérateur, la règle correcte peut être exprimée comme suit :

```
i %. → %. i
```

Recommandations.

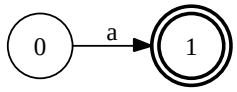
1. Cet opérateur doit être écrit **des deux côtés** de la règle.
2. Sections autorisées : seulement `RULE` (cf. Section 3.5).
3. Précédence : entre (4) et (5) de la table de précédence (cf. Section 1.3).

4.2 L'opérateur *complément*

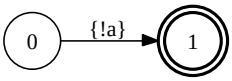
Le complément d'une expression régulière `REG` s'obtient comme ceci :

```
REG !
```

Difference entre l'opérateur *non* et l'opérateur *complément*. L'opérateur *non* (`[^abc]`) travaille sur des **symboles**. L'opérateur *complément* travaille sur des **langages** : il calcule la différence entre le monoïde libre Σ^* et le **langage** défini par l'expression `REG`.

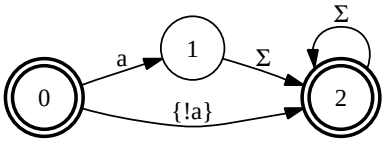


a



$[\hat{}a] = (\Sigma - a)$

non



$a! = (\Sigma^* - \{a\})$

complément

Recommandations.

1. Sections autorisées : classes, langages, règles et conditions.
2. Précédence : ligne (5) de la table de précédence (cf. Section 1.3).

4.3 L'opérateur *composition*

La composition de deux FSMs $\mathtt{T}_1$ et $\mathtt{T}_2$ s'obtient comme ceci :

$$\mathtt{T}_1 \ \circ \ \mathtt{T}_2$$

Recommandations.

1. Si les deux machines sont des automates, l'opération sera une **intersection**. Le point suivant (Section 4.4) vous donnera plus d'informations à ce sujet.
2. Sections autorisées : seulement `COMPILE` (cf. Section 3.8).
3. Précédence : ligne (6) de la table de précédence (cf. Section 1.3).

4.4 L'opérateur *projection*

Imaginez que vous ayez calculé un transducteur pondéré ou non, et que vous désiriez limiter ce transducteur à l'automate correspondant à son entrée ou à sa sortie. Dans ce cas, vous devez projeter l'entrée (première projection) ou la sortie (seconde projection) du transducteur.

Voici comment obtenir les première et seconde projections d'un transducteur `T` :

`T>i`	: première projection (entrée) d'un transducteur `T`
`T>o`	: seconde projection (sortie) d'un transducteur `T`

Recommandations.

1. Si vous voulez construire l'**intersection** de deux transducteurs,
   (a) **projetez** les deux transducteurs de manière à obtenir les automates désirés,
   (b) **composez** les automates obtenus ensemble.

   Par exemple, T_1 travaille sur les alphabets {`A:B`}, T_2 travaille sur les alphabets {`B:C`}. Il est possible de calculer l'intersection de la seconde projection de T_1 avec la première projection de T_2 comme suit :
   T_1`>o` $\circ$ T_2`>i`
2. Sections autorisées : seulement `COMPILE` (cf. Section 3.8).
3. Précédence : line (5) de la table de précédence (cf. Section 1.3).

5 Ligne de commande

En ligne de commande, lorsque vous tapez

```
./ovide
```

suivi du retour à la ligne, l'aide suivante apparaît :

```
ovide  mainfile.regexp [-t] [-g] [-r] [-c] [-q] [-mr|-bf] [-s··] [-d...] [-x...]
       [-min=N]

   -t          create fsmtxt
   -g          create graph
   -r          rebuild includes
   -c          classes in output FSM
   -q          quiet mode (no print)
   -mr|-bf     algorithms for compiling rules : mr=mohri (default),
               bf=beaufort
               Default is mohri
   -s··        sort on 2 features from [iows] (Input, Output, Weight,
               State)
               Default is "-siw" (input, weight)
   -d...       output directory is ... (relative or absolute)
   -x...       add suffix ... to output files
   -min=N      only for DICIN. The N value is the number of lines
               compiled before minimizing
               Default is 10,000
```

5.1 Comportement standard

Si le seul argument passé à Ovide est le nom d'un fichier Ovide (*e.g.* `mainfile.regexp`), ce fichier est compilé avec les options de compilation par défaut, et une seule sortie est produite : le FSM correspondant (`mainfile.fsm`). Les options par défaut sont les suivantes :

- Les fichiers de sortie sont créés dans le répertoire courant.
- *No rebuild* : les fichiers inclus ne sont pas recompilés si les FSMs correspondants existent.
- *No classes* : les transitions sont des symboles, pas des classes.
- L'algorithme classique de Mohri & Sproat (1996) est utilisé.
- Mode verbeux. L'étape en cours est imprimée sur l'écran.
- Les transitions des états sont ordonnées selon : (1) `Input`, (2) `Weight` (correspond à l'option "-siw" sur la ligne de commande).

5.2 Options

```
-t    create fsmtxt
```

Créer un FSM au format texte (extension `.fsmtxt`). Pour de plus amples informations concernant la syntaxe de ce type de fichier, consultez la documentation de la bibliothèque FSM.

```
-g    create graph
```

Créer une représentation graphique du FSM (extension `.graph`) qui peut être affichée à l'aide de l'outil *Graphviz* d'AT&T. Pour plus d'informations, voir `www.graphviz.org/`.

```
-r    rebuild includes
```

Si le fichier principal inclut d'autres fichiers, cette option oblige Ovide à recompiler les fichiers inclus. Dans le cas contraire, seul le fichier principal est recompilé.

```
-c    classes in output FSM
```

Afin de réduire au maximum la taille de la machine compilée, les transitions d'un même état p qui joignent un même état q avec le même poids w peuvent être combinées en une seule transition étiquetée par une classe de symboles.

Ovide intègre dans ce cas, dans la machine binaire, une hashmap contenant les classes détectées. Il faut noter qu'une machine qui présente des classes de symboles doit être pourvue de ses alphabets, afin de pouvoir subir la composition. Pour savoir comment ajouter des alphabets dans un FSM, consultez la documentation de la bibliothèque FSM.

```
-q    quiet mode (no print)
```

En mode par défaut, l'algorithme imprime un certain nombre d'informations sur la sortie standard : section en cours, opération en cours (composition, minimisation), etc. Le mode *quiet* vous permet de supprimer cet affichage...

```
-mr|-bf    algorithms for compiling rules
```

Dans Ovide, l'algorithme par défaut qui compile chaque règle séparément est celui de Mohri & Sproat (1996). Notez que vous pouvez malgré tout le spécifier sur la ligne de commande (`-mr`).

A la place de cet algorithme, vous pouvez également utiliser celui de Beaufort (2006) (`-bf`). Plus rapide, cet algorithme ne permet cependant pas encore de gérer des règles du type

$$a \rightarrow b : :c\Sigma^* \_ \Sigma^*d \tag{5.2.1}$$

où Σ^* inclut a lui-même... En position de Σ^*, a n'est pas réécrit avec ce nouvel algorithme.

```
-s··    sort on 2 features
```

Les transitions {entrée (`i`), sortie (`o`), poids (`w`), état suivant(`s`)} d'un état donné sont ordonnées par défaut sur l'entrée, et ensuite sur le poids (`-siw`), parce qu'un FSM compilé par Ovide sert généralement de second argument dans une composition.

L'option `-s` vous permet de spécifier les deux caractéristiques sur lesquelles les transitions des états de votre machine doivent être classées. Par exemple, `-sow` demande à Ovide un classement sur la sortie, et ensuite le poids, classement nécessaire à la première machine d'une composition.

Attention. Il ne faut **aucun espace** entre l'option et ses arguments.

```
-d...    output directoy
```

Cette option permet de spécifier un répertoire (relatif ou absolu) où doit être sauvé le résultat de la composition.

Attention. Il ne faut **aucun espace** entre l'option et ses arguments. Par exemple, si le répertoire est *example/graph/*, l'option sur la ligne de commande sera écrite `-dexample/graph/`. Notez le slash à la fin...

```
-x...    suffix
```

Le nom par défaut de la machine compilée est le nom du fichier Ovide chargé, dont l'extension `.regexp` est remplacée par l'extension `.fsm` : `syntax.regexp` → `syntax.fsm`.

L'option `-x` permet d'ajouter un suffixe au nom, avant l'extension. Par exemple, avec l'extension `-x_ext`, la sortie sera `syntax_ext.fsm`.

Attention. Il ne faut **aucun espace** entre l'option et son argument.

6 Exemples

Voici quelques exemples de fichiers Ovide avec leurs alphabets.

6.1 Un exemple simple

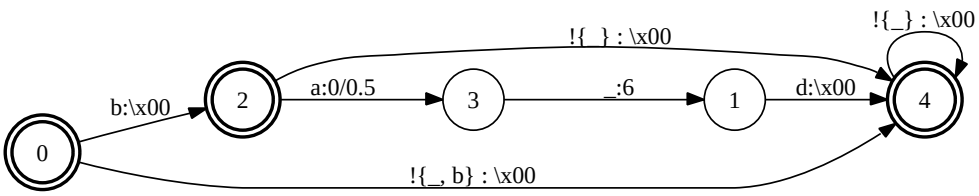
6.1.1 Le fichier Ovide

```
[INFO]
ALPHAIN  = alpha.symbol
ALPHAOUT = NUM

[RULE]
a?-> 06 :: ^b _ d / .5 # la règles est facultative...
```

6.1.2 alpha.symbol

```
\EPSILON=_
a
b
c
d
e
f
g
h
i
j
```

6.1.3 La machine compilée



6.2 Avec un fichier inclus

- Le fichier principal travaille sur les alphabets `cat.symbol` et `group.symbol` (A:B).
- Le fichier inclus travaille sur les alphabets `group.symbol` et `phrase.symbol` (B:C).
- L'interface des deux fichiers est donc `group.symbol` (B), et le résultat de la compilation sera un FSM travaillant sur `cat.symbol` et `phrase.symbol` (A:C).

6.2.1 syntax.regexp

```
[INFO]
ALPHAIN  = cat.symbol
ALPHAOUT = group.symbol

[RULE]
Noun -> N
Verb -> N :: Det _
Verb -> V

[INCLUDE]
syntax_inc
```

6.2.2 syntax_inc.regexp

```
[INFO]

ALPHAIN  = group.symbol
ALPHAOUT = phrase.symbol

[RULE]
V N -> VP
V -> VP
N -> NP
```

6.2.3 cat.symbol

```
\EPSILON=_
Det
Noun
Verb
```

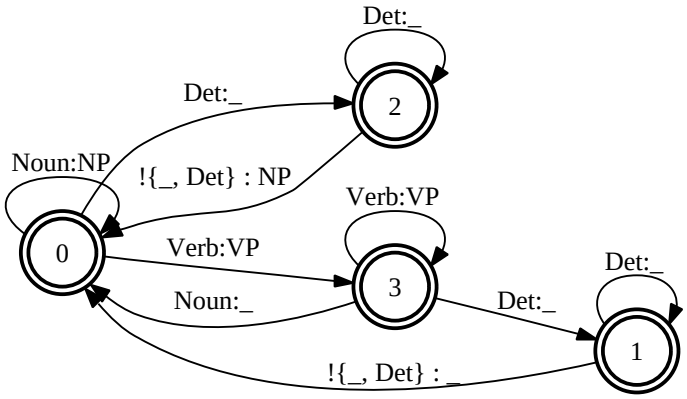
6.2.4 group.symbol

```
\EPSILON=_
N
V
```

6.2.5 phrase.symbol

```
\EPSILON=_
NP
VP
```

6.2.6 La machine compilée



6.3 Avec un filtre

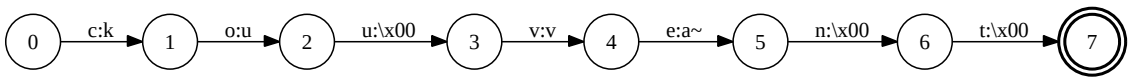
6.3.1 Le fichier Ovide

```
[INFO]
ALPHAIN  = ASCII
ALPHAOUT = Phonemes.symbol
FILTER   = filter.fsm

[DICIN]
couvent # le langage est réduit à "couvent"

[RULE]
couvent -> k u v a~ # doit être aligné avec le filtre
```

6.3.2 filter.regexp

```
[INFO]
ALPHAIN  = ASCII
ALPHAOUT = Phonemes.symbol

[RULE]
 c ?-> [ksS]
 c ?-> \x00 / 1  # c devient ϵ avec un poids de 1
 c -> . / 2      # c devient n'importe quoi d'autre avec un poids de 2
 o ?-> [Oo]
 o ?-> u / .5
 o ?-> \x00 / 1  # o devient ϵ avec un poids de 1
 o -> . / 2      # o devient n'importe quoi d'autre avec un poids de 2
 ...
```

6.3.3 La machine compilée



Annexe C

Sélection d'unités non uniformes

1 Les groupes rythmiques

Dans cette approche *chinks 'n chunks* (Liberman & Church 1991), le principe est de séparer les catégories entre la classe des *chinks* ou *mots-fonctions* (déterminant, préposition...) et celle des *chunks* ou *mots-contenus* (nom, verbe...). Un groupe, pour être valide, doit ensuite correspondre à l'expression suivante : (*chinks** *chunks**). En somme, dès qu'un *chunk* est suivi par un *chink*, on se trouve à une frontière de groupe.

Notre approche est un simple raffinement de cet algorithme. Nos catégories sont les suivantes :

1. *alone* : catégories qui constituent *toujours* un groupe à elles seules, comme les adverbes et pronoms interrogatifs.
2. *alwaysbegin* : catégories qui marquent toujours le début d'un nouveau groupe, comme une conjonction de coordination.
3. *head1* : têtes de groupe nominal.
4. *canbegin1* : catégories qui *peuvent* commencer un *head1*.
5. *sat1* : catégories satellites d'un *head1*, comme l'adjectif.
6. *head2* : têtes de groupe verbal.
7. *canbegin2* : catégories qui *peuvent* commencer un *head2*.
8. *sat2* : catégories satellites d'un *head2*, comme le pronom personnel complément.
9. *headqueue2* : catégories qui termine un *head2*, comme le participe passé.
10. *sat12* : catégories satellites d'un *head1* ou d'un *head2*, comme l'adverbe.

Sur la base de ces catégories, voici quelques exemples des groupes que nous acceptons :

1. *alone*
2. ((*alwaysbegin*| *canbegin1*) *sat1** *head1* ? *sat1** (*canbegin1 sat1* ? *head1* ? *sat1* ?) ?)

2 Exemple de table d'étiquetage

Les tables d'étiquetage des trois pages suivantes illustrent la phrase :

Novembre 1980 : bâtiment entre guillemets hors d'eau

où « 80 » est prononcé *huitante*.

Quelques remarques concernant ces tables :

1. Les tables se lisent en parallèle : une ligne de la table initiale a été scindée entre les 3 tables, afin de ne pas dépasser la largeur d'une page.
2. Afin de ne pas dépasser la longueur d'un page, certaines parties de la phrase ne sont pas illustrées. Dans les tables, ces parties sont entre parenthèses. Il s'agit des phonèmes correspondant à « mille neuf » (de *1980*) et « entre guille... » (de *entre guillemets*).
3. Dans les tables, les noms des critères sont abrégés :
   - Pho : phonème
   - Voi : voisement
   - T.A : type d'articulation
   - L.A : lieu d'articulation
   - M.A : mode d'articulation
   - Lab : labialité
   - Acc : accent
   - Syl : syllabe
   - Po.S : position dans la syllabe
   - Lg.S : longueur de la syllabe
   - Po.S.M : position de la syllabe dans le mot
   - Lg.M : longueur du mot
   - Po.S.G : position de la syllabe dans le groupe
   - G : groupe
   - Po.M.P : position du mot dans la phrase
   - Lg.P : longueur de la phrase
   - ID.P : numéro identifiant de la phrase
   - BEG : début du phonème (en échantillons)
   - END : fin du phonème (en échantillons)
   - F0 : fréquence fondamentale
   - E : énergie

Pho	**Voi**	**T.A**	**L.A**	**M.A**	**Lab**	**Acc**	**Syl**	**Po.S**
_	0	0	0	0	0	0	0	0
n	1	C	AL	NA	0	2	CV	ON
ɔ	1	V	VE	OR	A	2	CV	NU
v	1	C	LD	CS	0	1	CV	ON
ɑ̃	1	V	PA	NA	A	1	CV	NU
b	1	C	BI	PL	0	0	CV	ON
ʁ	1	C	UV	LI	0	0	CV	ON
ə	1	V	CE	OR	0	0	CV	NU
(mille neuf)								
s	0	C	AL	CS	0	1	CV	ON
ɑ̃	1	V	VE	NA	A	1	CV	NU
ɥ	1	S	PA	OR	A	2	V	ON
i	1	V	PA	OR	E	2	V	NU
t	0	C	DE	PL	0	1	CVC	ON
ɑ̃	1	V	VE	NA	A	1	CVC	NU
t	0	C	DE	PL	0	1	CVC	CO
n	1	C	DE	NA	0	1	CVC	ON
œ	1	V	PA	OR	A	1	CVC	NU
f	0	C	LD	CS	0	1	CVC	CO
_	0	0	0	0	0	0	0	0
b	1	C	BI	PL	0	2	CV	ON
a	1	V	PA	OR	A	2	CV	NU
t	0	C	DE	PL	0	0	CV	ON
i	1	V	PA	OR	E	0	CV	NU
m	1	C	BI	NA	0	1	CV	ON
ɑ̃	1	V	VE	NA	A	1	CV	NU
(entre guille...)								
m	1	C	BI	NA	0	1	CV	ON
ɛ	1	V	PA	OR	E	1	CV	NU
ɔ	1	V	VE	OR	A	0	VC	NU
ʁ	1	C	UV	LI	0	0	VC	CO
d	1	C	DE	PL	0	1	CV	ON
o	1	V	VE	OR	A	1	CV	NU
_	0	0	0	0	0	0	0	0

Table d'étiquetage de LiONS (1/3)

Pho	**Lg.S**	**Po.S.M**	**Lg.M**	**Po.S.G**	**G**	**Po.M.P**	**Lg.P**
_	0	0	0	0	0	BEG	MD
n	2	BEG	MD	BEG	HEAD1	BEG	MD
ɔ	2	BEG	MD	BEG	HEAD1	BEG	MD
v	2	IN	MD	IN	HEAD1	BEG	MD
ɑ̃	2	IN	MD	IN	HEAD1	BEG	MD
b	3	END	MD	END	HEAD1	BEG	MD
ʁ	3	END	MD	END	HEAD1	BEG	MD
ə	3	END	MD	END	HEAD1	BEG	MD
(mille neuf)							
s	2	BEG	SH	IN	HEAD1	IN	MD
ɑ̃	2	BEG	SH	IN	HEAD1	IN	MD
ɥ	2	BEG	SH	IN	HEAD1	IN	MD
i	2	BEG	SH	IN	HEAD1	IN	MD
t	3	END	SH	IN	HEAD1	IN	MD
ɑ̃	3	END	SH	IN	HEAD1	IN	MD
t	3	END	SH	IN	HEAD1	IN	MD
n	3	BEG	SH	END	HEAD1	IN	MD
œ	3	BEG	SH	END	HEAD1	IN	MD
f	3	BEG	SH	END	HEAD1	IN	MD
_	0	0	0	0	PUNCT	IN	MD
b	2	BEG	SH	BEG	HEAD1	IN	MD
a	2	BEG	SH	BEG	HEAD1	IN	MD
t	2	IN	SH	IN	HEAD1	IN	MD
i	2	IN	SH	IN	HEAD1	IN	MD
m	2	END	SH	IN	HEAD1	IN	MD
ɑ̃	2	END	SH	IN	HEAD1	IN	MD
(entre guille...)							
m	2	END	SH	IN	HEAD1	IN	MD
ɛ	2	END	SH	IN	HEAD1	IN	MD
ɔ	2	BEG	SH	IN	HEAD1	IN	MD
ʁ	2	BEG	SH	IN	HEAD1	IN	MD
d	2	BEG	SH	END	HEAD1	END	MD
o	2	BEG	SH	END	HEAD1	END	MD
_	0	0	0	0	PUNCT	END	MD

Table d'étiquetage de LiONS (2/3)

Pho	ID.P	BEG	END	F0	E
_	31	0	36048	0	9.529568
n	31	36048	37648	197	33.23319
ɔ	31	37648	39264	210	43.28882
v	31	39264	40592	168	37.53999
ɑ̃	31	40592	43392	200	38.32008
b	31	43392	43952	172	31.27890
ʁ	31	43952	44720	157	27.84247
ə	31	44720	45680	166	35.78213
(mille neuf)					
s	31	51728	53216	0	43.78566
ɑ̃	31	53216	55216	198	37.28046
ɥ	31	55216	56176	180	29.55567
i	31	56176	57136	163	34.31598
t	31	57136	58176	0	29.90840
ɑ̃	31	58176	59904	145	32.61069
t	31	59904	61312	0	30.58718
n	31	61312	62912	138	30.00321
œ	31	62912	65424	144	37.26070
f	31	65424	67616	0	33.12405
_	31	67616	72992	0	21.62800
b	31	72992	74624	152	18.98402
a	31	74624	75840	226	48.08130
t	31	75840	77568	0	36.89017
i	31	77568	78928	218	43.67994
m	31	78928	80368	194	30.91451
ɑ̃	31	80368	83024	211	35.25463
(entre guille...)					
m	31	92320	93712	243	29.49557
ɛ	31	93712	95344	247	41.69545
ɔ	31	95344	96976	241	36.69268
ʁ	31	96976	98240	178	31.30085
d	31	98240	99376	0	21.65087
o	31	99376	100928	130	25.59284
_	31	100928	125824	0	9.229073

Table d'étiquetage de LiONS (3/3)

3 LiONS 1 : pondération moyenne des critères du coût cible

La table ci-dessous présente la pondération moyenne (tous phonèmes confondus) pour les critères du coût cible, dans le cadre de la première version de LiONS. Selon le phonème, l'importance des caractéristiques varie, mais l'objectif de cette table est de mettre en évidence le fait que, en moyenne, certaines caractéristiques sont particulièrement mises en valeur par l'apprentissage. Dans la table,

1. Plus le poids est élevé, et plus une caractéristique est importante dans le calcul de la distance entre une cible et un candidat.
2. Les caractéristiques sont classées par ordre décroissant d'importance.
3. Une caractéristique concerne soit le phonème cible (C), soit le phonème de contexte gauche (G), soit le phonème de contexte droit (D).

Pho.	**Caractéristique**	**Poids**	**Pho.**	**Caractéristique**	**Poids**
C	Accent	35,000	G	Lèvres	3,380
G	Accent	33,997	D	Type syllabe	3,319
D	Accent	33,846	D	Nombre phonemes/syllabe	3,311
C	Nombre mots/phrase	25,234	D	Mode articulation	3,309
D	Position mot/phrase	12,436	G	Position mot/groupe	3,289
C	Position mot/phrase	11,656	C	Position mot/groupe	3,273
G	Position mot/phrase	10,940	G	Type syllabe	3,263
D	Type de phrase	4,603	G	Nombre phonemes/syllabe	3,180
C	Type de phrase	4,603	D	Position syllabe/mot	3,031
G	Type de phrase	4,595	G	Position syllabe/mot	3,028
C	Nombre phonemes/syllabe	4,325	D	Nombre syllabe mot	2,987
D	Voisement	4,104	C	Position syllabe/mot	2,932
D	Lèvres	3,951	G	Mode articulation	2,916
G	Type phonème (voyelle,...)	3,886	C	Nombre syllabe mot	2,911
C	Type syllabe	3,762	G	Nombre syllabe mot	2,853
G	Voisement	3,682	C	Type groupe rythmique	2,792
D	Type phonème (voyelle,...)	3,650	D	Type groupe rythmique	2,732
C	Position phoneme/syllabe	3,461	G	Type groupe rythmique	2,670
D	Position mot/groupe	3,390	G	Lieu articulation	2,493

Nous rappelons que cette pondération est le résultat d'un entraînement, réalisé par phonème, sur des ensembles de réalisations acoustiques obtenus par partitionnement, à l'aide de l'algorithme K-Means (McQueen 1967, Forgy 1965). L'ensemble des réflexions qui ont mené à l'établissement de la formule de pondération d'une caractéristique F_j :

$$W(F_j) = 2\ \log(1 + 10\, GR(F_j))$$

ont été détaillées en Section 10.1.5.

Annexe D

Correction orthographique

1 Catégories syntaxiques

Pour des précisions concernant les notions d'*unité linguistique* et de *nature*, nous renvoyons le lecteur à la Section 13.2.1.

1.1 Catégories valables pour les unités linguistiques et les natures

ACRONYM	: Acronyme
ADJ	: Adjectif
ADV	: Adverbe (sans classement particulier)
ADVDEG	: Adverbe de degré
ADVINT	: Adverbe d'interrogation
ADVN	: Adverbe de négation
AUX	: Verbe qui a la fonction d'auxiliaire
CONJ	: Conjonction (sans classement particulier)
CONJCOOR	: Conjonction de coordination
CONJSUB	: Conjonction de subordination
DET	: Déterminant (sans classement particulier)
DETEXCL	: Déterminant exclamatif
DETIND	: Déterminant indéfini
DETINT	: Déterminant interrogatif
DETPREP	: Préposition et déterminant fusionnés
EUPHO	: Euphonique
FOREIGN	: Mot étranger
GNUNDEFINED	: Mot de catégorie indéfinie
INFINIT	: Verbe à l'infinitif
INTERJ	: Interjection
INTMARK	: Ponctuation, marque d'interrogation

INTROD : Introducteur
INVARIABLE : Mot invariable (sans classement particulier)
NOUN : Nom commun
NUM : Nombre
PARTPASSE : Verbe au participe passé
PARTPRES : Verbe au participe présent
PAUSE : Pause
PRED : Verbe copule, qui précède un prédicat
PREF : Préfixe
PREP : Préposition
PRON : Pronom (sans classement particulier)
PRONDEM : Pronom démonstratif
PRONIND : Pronom indéfini
PRONINT : Pronom interrogatif
PRONPER : Pronom personnel de fonction indéfinie
PRONPERCD : Pronom personnel complément direct
PRONPERCI : Pronom personnel complément indirect
PRONPERSJ : Pronom personnel sujet
PRONPOS : Pronom possessif
PRONREL : Pronom relatif
PROPERNAME : Nom propre
SYMBOL : Symbole
TIRET : Tiret
VERB : Verbe (sans classement particulier)
ENDPUNCT : Punctuation finale
LIGHTPUNCT : Punctuation légère
MEDIUMPUNCT : Punctuation moyenne
GUILLEMET : Guillemet (ouvrant ou fermant)
PAROUV : Parenthèse ouvrante
PARFER : Parenthèse fermante
EXCLMARK : Ponctuation, marque d'exclamation

1.2 Catégories valables exclusivement pour les unités linguistiques

FIELD : Champ d'information
GNDATE : Date
MONEY : Monnaie, devise
TEL : Téléphone
TIME : Temps
UNIT : Unité

URI : Identificateur uniforme de ressource Internet [1]
EOS : Fin de phrase [2]

1.3 Réflexions qui ont mené à la constitution de cette liste de catégories

> « On distingue généralement deux catégories de mots : les *mots grammaticaux* (déterminants, pronoms, prépositions, conjonctions), en nombre fini (moins de 1000), et les *mots lexicaux*, en nombre a priori infini. [...] Les mots grammaticaux forment le squelette syntaxique de la phrase. Leur identification est fondamentale. » (Boîte *et al.* 2000, p. 352)

Nous fondant sur ce principe, nous avons décidé de distinguer les mots grammaticaux tant que faire se peut. Nous avons de ce fait subdivisé les catégories DET (DETIND, DETEXCL, etc.) et PRON (PRONDEM, PRONPER, etc.), dont le comportement syntaxique dépend fortement du type.

Notons que parmi la catégorie ADV, qui contient des mots lexicaux, il est également possible de distinguer certaines sous-catégories relativement fermées et dont le comportement syntaxique s'apparente à celui des mots grammaticaux. C'est ainsi que nous avons ajouté les sous-catégories ADVDEG (plus, moins, etc.), ADVINT (comment, quand, etc.) et ADVN (ne, pas, plus, etc.).

Nous détaillons ci-dessous l'intérêt de trois catégories fort discriminantes parmi les pronoms : les pronoms possessifs, démonstratifs et personnels.

Pronoms possessifs. L'intérêt de cette catégorie vient du fait que seuls ces pronoms admettent d'être précédés par un déterminant : « le mien », « les nôtres ».

Pronoms démonstratifs. En position d'antécédent d'un relatif, les pronoms de cette catégorie sont les plus probables. D'autres pronoms apparaissent parfois, mais beaucoup plus rarement à cette position.

Pronoms personnels. Les catégories PRONPERSJ, PRONPERCD et PRONPERCI ont été ajoutées, pour distinguer des pronoms personnels qui peuvent également être étiquetés déterminant (le, la, les, leur) ou participe passé (tu), mais qui n'acceptent qu'une seule analyse en tant que pronom personnel.

Distinguer ces différents types permet d'affiner les connaissances quant à leur comportement. Par exemple, un PRONPERCD a peu de probabilité d'apparaître en début de phrase, au contraire d'un PRONPERSJ. Ces distinctions aident l'analyseur à choisir la bonne analyse. Sur des phrases courtes (*Le couvent. La ferme.*), seule cette distinction permet de choisir le déterminant au lieu du pronom.

[1] URI : *Uniform Resource Identifier*. Il s'agit d'une chaîne de caractères identifiant une ressource Web (physique ou abstraite) dont la syntaxe respecte une norme W3C. Voir www.w3.org/Addressing/.
[2] EOS : *End Of Sentence*.

Les pronoms qui appartiennent à la catégorie générale PRONPER ne peuvent être que pronom personnel. Ils n'ont donc pas été distingués. La table ci-dessous donne la liste complète des différentes sous-catégories créées.

PRONPER	
moi, toi	: disjoint non réfléchi / réfléchi
me, m', te, t'	: conjoint CD / CI / réfléchi
lui	: conjoint CI / disjoint non réfléchi
elle, elles	: conjoint sujet / disjoint non réfléchi
nous, vous	: toutes les fonctions
PRONPERSJ	
je, j'	: 1, sg
tu	: 2, sg
il, on	: 3, sg
ils	: 3, pl
PRONPERCD	
le, la, l'	: 3, sg
les	: 3, pl
PRONPERCI	
leur	: 3, pl
en, y	: indéfini

2 Composés à traits d'union

2.1 2 parties

Mots	Unité linguistique
PREF - ADJ PREP - ADJ	ADJ
NOUN - NOUN PREP - NOUN ADJ - NOUN ADV - ADJ DETPREP - ADV NUM - NOUN PREF - NOUN	NOUN
ADJ - VERB NOUN - VERB PREP - VERB VERB - VERB	VERB
ADJ - INFINIT NOUN - INFINIT PREP - INFINIT VERB - INFINIT	INFINIT
PREP - PARTPRES	PARTPRES
PREP - PARTPASSE	PARTPASSE
DETIND - PRONIND	PRONIND
NOUN - PROPERNAME PROPERNAME - NOUN	PROPERNAME

2.2 3 parties

Mots	Unité linguistique
PREF - PREP - NOUN	ADJ
ADJ - ADJ - NOUN VERB - PREP - NOUN	NOUN

3 Traits grammaticaux et classes flexionnelles

3.1 Traits grammaticaux

La table ci-dessous présente les traits grammaticaux utilisés dans la nouvelle analyse morpho-syntaxique d'eLite.

Trait	Valeur	Signification
Genre	`m`	masculin
	`f`	féminin
	`GND`	genre indéfini
Nombre	`s`	singulier
	`p`	pluriel
	`NND`	nombre indéfini
Personne	`1st`	1^{re} personne
	`2nd`	2^{e} personne
	`3rd`	3^{e} personne
	`PND`	personne indéfinie

3.2 Classes flexionnelles

3.2.1 Verbes

Les classes flexionnelles verbales sont principalement identifiées par des indices allant de 1 à 84. Certaines classes sont cependant complétées d'une lettre (*a*, *b*, *c*, ...), lorsque seules des petites variations distinguent deux classes fort proches [3]. Voici quelques classes remarquables :

1. Classe 1 : *avoir*.
2. Classe 2 : *être*.
3. Classe 6 : conjugaison des verbes en *-er*.
4. Classe 19a : conjugaison des verbes inchoatifs (*-ir*, *-issant*).

3.2.2 Noms et adjectifs

Les classes flexionnelles des noms et des adjectifs sont plus explicites. L'identifiant d'une classe flexionnelle combine les traits grammaticaux et les suffixes flexionnels qui varient dans la classe :

1. Les différentes valeurs qui identifient la classe sont séparées par des virgules.
2. Les traits utilisés sont le genre et le nombre.

[3] Par exemple, 7*a* et 7*b*, pour les classes des verbes respectivement en *-cer* et *-ecer*

3. La classe indique, pour le genre et pour le nombre, quel suffixe doit être remplacé pour transformer la forme fléchie en lemme, sous la forme « `-`*A*`+`*B* », où *A* est le suffixe à remplacer, et *B* est le suffixe canonique du lemme.
4. Quelques exemples :
    - `mf,sp,-s+,-èche+ec` est la classe flexionnelle des noms qui varient en genre et en nombre, dont le lemme est reconstitué en supprimant *-s* au pluriel, et en remplaçant *-èche* par *-ec* au féminin.
    - `m,sp,-aux+ail,-+` est la classe flexionnelle des noms masculins, qui varient en nombre, et dont le lemme est reconstitué en remplaçant *-aux* par *ail* lorsque la forme est au pluriel.

3.2.3 Autres catégories

Les classes flexionnelles des autres catégories indiquent le nom de la catégorie concernée. Par exemple, l'article défini est identifié par la classe *det_art*, et le pronom possessif, par la classe *det_poss*.

4 Pseudocodes de l'analyse morpho-syntaxique

L'algorithme général de l'analyse morpho-syntaxique est présenté en Pseudocode 41. Les différents traitements réalisés sont initialisés par cet algorithme et sont présentés dans les pseudocodes suivants. Le véritable processus de correction commence avec le Pseudocode 42 qui réalise l'analyse des tokens lexicaux.

Require: DLS, la structure de données
Ensure: L'analyse morpho-syntaxique des phrases de la DLS est réalisée, les unités linguistiques sont créées

1: $FirstWord \leftarrow$ `GetFirstWord`(DLS)
2: $NbWord \leftarrow$ `CountWordInCurSent`$(DLS, FirstWord)$
3: $FSMVec \leftarrow$ `InitFSMVec`$(NbWord)$
4: $NbFSM \leftarrow 0$
5: **while** `GetCurWord`$(DLS) \neq$ `NULL` **do**
6: $Type \leftarrow$ `GetTypeCurToken`(DLS)
7: **if** $Type =$ `MOT` **then**
8: $FSMVec[NbFSM] \leftarrow$ `ProcessLex`(DLS)
9: $NbFSM \leftarrow NbFSM + 1$
10: **else if** $Type =$ `URI` **then**
11: $FSMVec[NbFSM] \leftarrow$ `ProcessURI`(DLS)
12: $NbFSM \leftarrow NbFSM + 1$
13: **else**
14: $FSMVec[NbFSM] \leftarrow$ `ProcessOtherToken`$(DLS, Type)$
15: $NbFSM \leftarrow NbFSM + 1$
16: **if** $Type =$ `ENDPUNCT` **then**
17: $SyntaxFSM \leftarrow$ `SyntacticAnalysis`$(FSMVec, NbFSM)$
18: $SentFSM \leftarrow$ `FlexionAnalysis`$(SyntaxFSM, NbFSM)$
19: `SaveInDLS`$(DLS, SentFSM, FirstWord)$
20: $FirstWord \leftarrow$ `GetNextWord`(DLS)
21: $NbWord \leftarrow$ `CountWordInCurSent`$(DLS, FirstWord)$
22: `Free`$(FSMVec)$
23: $FSMVec \leftarrow$ `InitFSMVec`$(NbWord)$
24: $NbFSM \leftarrow 0$
25: **end if**
26: **end if**
27: `SetCurWord`$(DLS,$ `GetNextWord`$(DLS))$
28: **end while**

Pseudocode 41: `MorphoSyntax`

Require: DLS, la structure de données
Ensure: L'analyse flexionnelle des words de type `MOT`contigus est réalisée
1: $FirstWord \leftarrow$ `GetCurWord`(DLS)
2: $LastWord \leftarrow$ `SearchLastWord`$(DLS, FirstWord)$
3: $WordGroupFSM \leftarrow$ `CreateFSM`$(DLS, FirstWord, LastWord)$
4: $MatchResFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, MatchFSM)$
5: `FSM_Project`$(MatchResFSM,$ `OUT`$)$
6: $WordCompResFSM \leftarrow$ `FSM_Compose`$(MatchResFSM, WordCompFSM)$
7: `FSM_Minimize`$(WordCompResFSM)$
8: $BestFSM \leftarrow$ `FSM_Prune`$(WordCompResFSM, 0.)$
9: `FSM_Balance`$(BestFSM, 0.)$
10: $(FSMVec, TypeVec, NbPart) \leftarrow$ `SplitOnLexCompound`$(BestFSM)$
11: **for** $i \leftarrow 0$ to $NbPart$ **do**
12: **if** $TypeVec[i] =$ `COMPOUND` **then**
13: `ProcessCompoundLex`$(DLS, FSMVec[i])$
14: **else**
15: `ProcessStdLex`$(DLS, FSMVec[i])$
16: **end if**
17: **end for**
18: $WordGroupFSM \leftarrow$ `ConcatAllFSMWithSpace`$(FSMVec, NbPart)$
19: `SetCurWord`$(DLS,$ `GetPrevWord`$(DLS))$
20: **return** $WordGroupFSM$

Pseudocode 42: `ProcessLex`

Require: DLS, la structure de données
$WordGroupFSM$, un mot composé sans analyse
Ensure: $WordGroupFSM$ contient le mot composé et son analyse
1: $CompNatResFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, WordCompNatFSM)$
2: `FSM_Project`$(CompNatResFSM,$ `OUT`$)$
3: $WordGroupFSM \leftarrow$ `FSM_Compose`$(CompNatResFSM, WordCompUnitFSM)$
4: $WordC \leftarrow$ `UpdateDLS`$(WordGroupFSM)$
5: `SetCurWord`$(DLS, WordC)$

Pseudocode 43: `ProcessCompoundLex`

Require: DLS, la structure de données
$WordGroupFSM$, un groupe de mots sans analyse
Ensure: $WordGroupFSM$ est analysé, et d'éventuels mots composés sont détectés sur les natures
1: $MatchResFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, MatchFSM)$
2: $OriginResFSM \leftarrow$ `FSM_Compose`$(MatchResFSM, OriginFSM)$
3: $(FSMVec, OriginVec, NbPart) \leftarrow$ `SplitOnOOV`$(DLS, OriginResFSM)$
4: **for** $i \leftarrow 0$ to $NbPart$ **do**
5: **if** $OriginVec[i] =$ `OOV` **then**
6: $OriginVec[i] \leftarrow$ `ProcessOOV`$(FSMVec[i], OriginVec[i])$
7: **else**
8: `ProcessIV`$(FSMVec[i])$
9: **end if**
10: **end for**
11: $WordGroupFSM \leftarrow$ `ConcatAllFSMWithSpace`$(FSMVec, NbPart)$
12: $NatureFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, KeepNatureOnlyFSM)$
/* Création éventuelle de composés sur la base des nature */
13: $CompTestFSM \leftarrow$ `FSM_Copy`$(NatureFSM)$
14: `FSM_Project`$(CompTestFSM,$ `OUT`$)$
15: `FSM_Balance`$(CompTestFSM, 0.)$
16: `FSM_Minimize`$(CompTestFSM)$
17: $RulesCompResFSM \leftarrow$ `FSM_Compose`$(CompTestFSM, RulesCompFSM)$
18: `FSM_Minimize`$(RulesCompResFSM)$
19: `FSM_Prune`$(RulesCompResFSM, 0.)$
20: $WordGroupFSM \leftarrow$ `FSM_Compose`$(NatureFSM, RulesCompResFSM)$
21: `FSM_Minimize`$(WordGroupFSM)$

Pseudocode 44: `ProcessStdLex`

Require: $WordGroupFSM$, un groupe de mots IV
Ensure: Les mots de $WordGroupFSM$ sont analysés et pondérés et au besoin, complétés de nouvelles flexions
1: $LexiconResFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, LexiconFSM)$
2: $WordGroupFSM \leftarrow LexiconResFSM$
3: `FSM_Project`$(WordGroupFSM,$ `OUT`$)$

Pseudocode 45: `ProcessIV`

Require: DLS, la structure de données
$WordFSM$, un mot OOV
Ensure: $WordFSM$ est analysé, et peut-être devenu IV
1: $OOVResFSM \leftarrow$ `FSM_Compose`$(WordFSM, UpperFSM)$
2: **if** $OOVResFSM =$ `NULL` **then**
3: $Res1FSM \leftarrow$ `FSM_Compose`$(WordFSM, OOVKeyboardLimitFSM)$
4: **if** $Res1FSM \neq$ `NULL` **then**
5: $TmpFSM \leftarrow$ `FSM_Compose`$(Res1FSM, OOVKeyboardFSM)$
6: **if** $TmpFSM \neq$ `NULL` **then**
7: `FSM_Project`$(TmpFSM,$ `OUT`$)$
8: $Res1FSM \leftarrow$ `FSM_Compose`$(TmpFSM, LexiconFSM)$
9: `FSM_Project`$(Res1FSM,$ `IN`$)$ **if** $Res1FSM \neq$ `NULL`
10: **end if**
11: **end if**
12: $TmpFSM \leftarrow$ `FSM_Compose`$(WordFSM, OOVPhoneFSM)$
13: **if** $TmpFSM \neq$ `NULL` **then**
14: `FSM_Project`$(TmpFSM,$ `OUT`$)$
15: $Res2FSM \leftarrow$ `FSM_Compose`$(TmpFSM, LexiconFSM)$
16: `FSM_Project`$(TmpFSM2,$ `IN`$)$ **if** $Res2FSM \neq$ `NULL`
17: **end if**
18: **if** $Res1FSM =$ `NULL` **and** $Res2FSM =$ `NULL` **then**
19: $OOVResFSM \leftarrow$ `FSM_Compose`$(WordFSM, OOVDerivFSM)$
20: **else**
21: $Origin \leftarrow IV$
22: **if** $Res1FSM \neq$ `NULL` **and** $Res2FSM \neq$ `NULL` **then**
23: $OOVResFSM \leftarrow$ `FSM_Minimize`(`FSM_Union`$(Res1FSM, Res2FSM)$)
24: **else if** $Res1FSM \neq$ `NULL` **then**
25: $OOVResFSM \leftarrow Res1FSM$
26: **else**
27: $OOVResFSM \leftarrow Res2FSM$
28: **end if**
29: **end if**
30: **end if**
31: $WordFSM \leftarrow OOVResFSM$
32: `FSM_Project`$(WordFSM,$ `OUT`$)$
33: **return** $Origin$

Pseudocode 46: `ProcessOOV`

Require: DLS, la structure de données
Ensure: L'analyse flexionnelle des words qui appartiennent au même token de type URI
1: $FirstWord \leftarrow$ `GetCurWord`(DLS)
2: $LastWord \leftarrow$ `SearchLastTokenWord`$(DLS, FirstWord)$
3: $URIGroupFSM \leftarrow$ `CreateFSM`$(DLS, FirstWord, LastWord)$
4: $URIResFSM \leftarrow$ `FSM_Compose`$(URIGroupFSM, TokenURIFSM)$
5: `FSM_Project`$(URIResFSM,$ `OUT`$)$
6: $(FSMVec, KeywordVec, NbPart) \leftarrow$ `SplitOnKeyword`$(DLS, URIResFSM)$
7: **for** $i \leftarrow 0$ to $NbPart$ **do**
8: **if** $KeywordVec[i] =$ `KEYWORD` **then**
9: `ProcessKeywordUri`$(DLS, FSMVec[i])$
10: **else**
11: `ProcessLexUri`$(DLS, FSMVec[i])$
12: **end if**
13: **end for**
14: $URIGroupFSM \leftarrow$ `ConcatAllFSMWithSpace`$(FSMVec, NbPart)$
15: `SetUnitOnOutput`$(URIGroupFSM,$ `URI`$)$
16: `SetCurWord`$(DLS,$ `GetPrevWord`$(DLS))$
17: **return** $URIGroupFSM$

Pseudocode 47: `ProcessURI`

Require: DLS, la structure de données
$WordFSM$, un mot d'une URI qui n'appartient pas aux mots-clefs
Ensure: L'analyse flexionnelle du mot, soit comme une suite de mots connus, soit comme un seul OOV
1: $MatchResFSM \leftarrow$ `FSM_Compose`$(WordFSM, MatchFSM)$
2: `FSM_Project`$(MatchResFSM,$ `OUT`$)$
3: $URIPartResFSM \leftarrow$ `FSM_Compose`$(MatchResFSM, URIPartFSM)$
4: **if** $URIPartResFSM \neq$ `NULL` **then**
5: `FSM_Minimize`$(URIPartResFSM)$
6: $WordFSM \leftarrow$ `FSM_GetBestPath`$(URIPartResFSM, 1)$
7: `ProcessIVForURI`$(DLS, WordFSM)$
8: **else**
9: $WordFSM \leftarrow$ `FSM_GetBestPath`$(MatchResFSM, 1)$
10: `ProcessOOVForURI`$(WordFSM)$
11: `SetCurWord`$(DLS,$ `GetNextWord`$(DLS))$
12: **end if**

Pseudocode 48: `ProcessLexURI`

Require: DLS, la structure de données
$WordFSM$, un mot ou une suite de mots IV dans une partie d'URI
Ensure: $WordFSM$ contient le mot ou la suite de mots et leur analyse
1: $WordGroupFSM \leftarrow$ `FSM_Compose`$(WordFSM, IV2SpaceFSM)$
2: `FSM_Project`$(WordGroupFSM,$ `OUT`$)$
3: $WordFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, LexiconFSM)$
4: `FSM_Project`$(WordFSM,$ `OUT`$)$
5: `SplitURIWordInDLS`$(DLS, WordFSM)$
6: `SetCurWord`$(DLS,$ `GetNextWord`$(DLS))$

Pseudocode 49: `ProcessIVForURI`

Require: $WordFSM$, une forme OOV dans une partie d'URI
Ensure: $WordFSM$ contient les analyses pondérées qui lui sont attribuables
1: $OOVResFSM \leftarrow$ `FSM_Compose`$(WordFSM, OOVDerivFSM)$
2: $WordFSM \leftarrow OOVResFSM$
3: `FSM_Project`$(WordFSM,$ `OUT`$)$

Pseudocode 50: `ProcessOOVForURI`

Require: DLS, la structure de données
$Type$, le type du token (tout sauf MOT et URI)
Ensure: $TokenResFSM$, un FSM contenant l'analyse du token de type $Type$
1: $FirstWord \leftarrow$ `GetCurWord`(DLS)
2: $LastWord \leftarrow$ `SearchLastTokenWord`$(DLS, FirstWord)$
3: $WordGroupFSM \leftarrow$ `CreateFSM`$(DLS, FirstWord, LastWord)$
4: $TokenFSM \leftarrow$ `ChooseLexiconFSM`$(Type)$
5: $UnitSymbol \leftarrow$ `ChooseUnitSymbol`$(Type)$
6: $TokenResFSM \leftarrow$ `FSM_Compose`$(WordGroupFSM, TokenFSM)$
7: `FSM_Project`$(TokenResFSM,$ `OUT`$)$
8: `SetUnitOnOutput`$(TokenResFSM, UnitSymbol)$
9: **return** $TokenResFSM$

Pseudocode 51: `ProcessOtherToken`

Require: $FSMVec$, un vecteur de FSMs correspondant à une phrase de la DLS
$NbFSM$, le nombre de FSMs dans le vecteur $FSMVec$
Ensure: $BestFSM$, un FSM contenant la meilleure analyse pour la totalité de la phrase
1: $AllFSM \leftarrow$ `ConcatAllFSMWithSpace`$(FSMVec, NbFSM)$
2: $CatFSM \leftarrow$(`FSM_Concat`$(AllFSM, EosFSM)$)
3: `FSM_Minimize`(`FSM_Project`$(CatFSM,$ `OUT`$)$)
4: $CatResFSM \leftarrow$ `FSM_Compose`$(CatFSM, CatNgramFSM)$
5: $SyntagmResFSM =$ `FSM_Compose`$(CatResFSM, SyntagmFSM)$
6: `FSM_Project`$(SyntagmResFSM,$ `IN`$)$
7: $SentFSM \leftarrow$ `FSM_Compose`$(AllFSM, SyntagmResFSM)$
8: $BestFSM \leftarrow$ `FSM_GetBestPath`$(SentFSM, 1)$
9: **return** $BestFSM$

Pseudocode 52: `SyntacticAnalysis`

Require: $SentFSM$, un FSM correspondant à une phrase de la DLS
$NbFSM$, le nombre de FSMs dans le vecteur $FSMVec$
Ensure: L'analyse flexionnelle est réalisée
1: $FSMVec$`SplitOnTokenLex`$(SentFSM, NbFSM)$
2: **for** $i \leftarrow 0$ to $NbPart$ **do**
3: **if** $TokenLexVec[i] =$ `TRUE` **then**
4: $FSMVec[i] \leftarrow FlexionProcess(FSMVec[i])$
5: **end if**
6: **end for**
7: $SentFSM \leftarrow$ `ConcatAllFSMWithSpace`$(FSMVec, NbFSM)$

Pseudocode 53: `FlexionAnalysis`

Require: $PartSentFSM$, un vecteur de FSMs correspondant à une suite de tokens lexicaux
Ensure: $BestFSM$, un FSM contenant la meilleure analyse flexionnelle correspondante
1: $FlexFSM \leftarrow$ `FSM_Copy`$(PartSentFSM)$
2: `FSM_Project`$(FlexFSM,$ `IN`$)$
3: $DetResFSM \leftarrow$ `FSM_Compose`$(FlexFSM, DetFlexFSM)$
4: $GenResFSM \leftarrow$ `FSM_Compose`$(DetResFSM, GenFlexFSM)$
5: $FlexResFSM \leftarrow$ `FSM_Compose`$(GenResFSM, FlexNgramFSM)$
6: $FlexFSM \leftarrow$ `FSM_Compose`$(FlexResFSM, PartSentFSM)$
7: $BestFSM \leftarrow$ `FSM_GetBestPath`$(FlexFSM, 1)$
8: **return** $BestFSM$

Pseudocode 54: `FlexionProcess`

Require: DLS, la structure de données
$SentFSM$, le FSM contenant l'analyse complète de la phrase
$FirstWord$, le pointeur sur le premier mot de la phrase dans la DLS
Ensure: Remplit la DLS de toutes les informations utiles à la suite du traitement : orthographe et nature de chaque Word, et création des unités linguistiques correspondantes

1: $CurWord \leftarrow$ `GetCurWord`(DLS)
2: `SetCurWord`($FirstWord$)
3: $Trans \leftarrow$ `FSM_GetAllTrans`($SentFSM$, `FSM_GetInitial`($SentFSM$))
4: $Symbol1 \leftarrow$ `TransFnd_GetIn`($Trans, 0$)
5: $Spelling \leftarrow$ `NULL`
6: **while** $Symbol1 \neq$ `EOS` **do**
7: **if** `IsAscii`($Symbol1$) = `TRUE` **and** `IsSpace`($Symbol1$) = `FALSE` **then**
8: `StringConcat`($Spelling, Symbol1$)
9: **else if** `IsCat`($Symbol1$) = `TRUE` **then**
10: $RemoveResFSM \leftarrow$ `NULL`
11: **if** $GetCurToken(DLS) =$ `MOT` **then**
12: $CatFSM =$ `FSM_CreateVal`($Symbol1$)
13: $RemoveResFSM \leftarrow$ `FSM_Compose`($CatFSM, RemoveListFSM$)
14: **if** $RemoveResFSM \neq$ `NULL` **then**
15: `RemoveCurWord`(DLS)
16: **end if**
17: **end if**
18: **if** $RemoveResFSM =$ `NULL` **then**
19: `SetCurWordValue`($DLS, Spelling, Symbol1$)
20: $Symbol2 \leftarrow$ `TransFnd_GetOut`($Trans, 0$)
21: **if** $Symbol2 \neq$ `EPSILON` **then**
22: `CreateUnitForCurWord`($DLS, Symbol2$)
23: **else**
24: `LinkCurWordToCurUnit`(DLS)
25: **end if**
26: **end if**
27: $Spelling \leftarrow$ `NULL`
28: `SetCurWord`(DLS, `GetNextWord`(DLS))
29: **end if**
30: $StateN \leftarrow$ `TransFnd_GetNext`($Trans, 0$)
31: $Trans \leftarrow$ `FSM_GetAllTrans`($SentFSM, StateN$)
32: $Symbol1 \leftarrow$ `TransFnd_GetIn`($Trans, 0$)
33: **end while**
34: `SetCurWord`($CurWord$)

Pseudocode 55: `SaveInDLS`

5 Segmentation dans le cas de la dichotomie par alignement

Require: I_D, un FSM à segmenter
$TypeNoSplit$, la valeur pour laquelle les données restent dans le même FSM
Ensure: $(FSMVec, TypeVec, NbPart)$, qui contiennent respectivement la segmentation, les types et le nombre de parties

1: $NbPart \leftarrow 0$
2: $start \leftarrow stateC \leftarrow 0$
3: $Trans \leftarrow$ `FSM_GetFirstTrans`$(I_D, 0)$
4: $Type \leftarrow Trans.Out$
5: **while** `FSM_IsFinal`$(stateC) =$ `FALSE` **do**
6: **if** $stateC \neq 0$ **and** $Trans.Out \neq$ `EPSILON` **then**
7: **if** $Trans.Out \neq TypeNoSplit$ **or** $(Type \neq TypeNoSplit)$ **then**
8: $FSMVec[NbPart] \leftarrow$ `FSM_CopyPart`$(I_D, start, stateC-1)$ `# Espace pas copié`
9: `FSM_Project`$(FSMVec[NbPart],$ `IN`$)$
10: $TypeVec[NbPart] \leftarrow Type$
11: $NbPart \leftarrow NbPart+1$
12: $start \leftarrow stateC$
13: $Type \leftarrow Trans.Out$
14: **end if**
15: **else if** `FSM_IsFinal`$(Trans.Next) =$ `TRUE` **then**
16: $FSMVec[NbPart] \leftarrow$ `FSM_CopyPart`$(I_D, start, Trans.Next)$
17: `FSM_Project`$(FSMVec[NbPart],$ `IN`$)$
18: $TypeVec[NbPart] \leftarrow Type$
19: $NbPart \leftarrow NbPart+1$
20: **end if**
21: $stateC \leftarrow Trans.Next$
22: $Trans \leftarrow$ `FSM_GetFirstTrans`$(I_D, stateC)$
23: **end while**
24: **return** $(NbPart \neq 0)$? $(FSMVec, TypeVec, NbPart)$: 0

Pseudocode 56: Segmentation dans le cas d'une dichotomie par alignement

Annexe E

Postulats et hypothèses

Les travaux réalisés dans cette thèse reposent sur un ensemble de postulats et d'hypothèses [1], que nous rassemblons dans cette annexe afin de donner au lecteur intéressé une aperçu général des directions qui ont été prises. Les justifications et les analyses, quant à elles, ont été données dans le corps du document, lorsque l'hypothèse ou le postulat concernés interviennent.

1 Démarche globale

Postulat 1 (Diviser pour mieux régner). *Tout comme un algorithme est une succession d'étapes simples, une tâche complexe n'est bien souvent qu'un ensemble de tâches simples, mais de natures fort différentes. L'identification de ces tâches simples permet dès lors de leur appliquer des solutions simples, inconcevables dans le contexte de la tâche complexe. Il s'agit donc de diviser pour mieux régner.*

Hypothèse 1 (Diviser pour mieux régner). *Pour autant qu'une tâche complexe puisse être analysée comme une succession d'étapes simples, les machines à états finis constituent l'outil idéal pour représenter cette succession d'étapes simples de manière aisée.*

Postulat 2 (Externalisation des données). *En traitement de la langue, une application qui se veut multilingue doit veiller à respecter une séparation stricte entre l'algorithme et les données dépendant de la langue.*

Hypothèse 2 (Externalisation des données). *Pour autant que les langues traitées partagent suffisamment de similarités, les machines à états finis autorisent l'externalisation de l'ensemble des traitements dépendant de la langue. Tout l'art réside dès lors dans la détection, au sein d'un processus, des traitements qui ressortissent à la langue.*

[1] Les termes *postulat* et *hypothèse* ont été définis dans l'introduction (Section 3, page 6).

2 Sélection d'unités non uniformes

2.1 Postulats linguistiques

Postulat 10.1.1 (Unité de base). *La seule unité qui autorise un traitement du signal limité au point de concaténation est le diphone, étant donné que dans tous les cas, la concaténation est réalisée en partie stable du signal. L'apparente impossibilité de couvrir le diphone à l'aide d'un corpus de parole est simplement le reflet de la richesse réelle de la langue, que des unités non contextuelles comme le phone ou le demi-phone ne modélisent pas correctement.*

Postulat 10.1.2 (Accent de mot). *En français, l'accent de mot est conservé, mais s'atténue par rapport à l'accent de groupe. Deux* degrés *d'accentuation existent donc en français : un degré* majeur *au niveau du groupe et un degré* mineur *au niveau du mot.*

2.2 Principes de sélection

Hypothèse 10.1.1 (Critères de sélection). *Un système de synthèse par sélection peut produire de la parole de haute qualité en n'utilisant* que *des informations linguistiques segmentales et suprasegmentales au niveau du coût cible, pour autant que le coût de concaténation soit* exclusivement *dirigé par des critères acoustiques.*

Hypothèse 10.1.2 (Pondération). *Les critères linguistiques du coût cible dirigent une prise de décision prosodique. Ils sont donc de nature à être départagés par entropie.*

Hypothèse 10.1.3 (Pondération par phonème). *Du fait de leurs différences articulatoires, les phonèmes se comportent différemment les uns des autres dans un même contexte d'élocution. De ce fait, une seule pondération des caractéristiques linguistiques ne serait pas pertinente. Il est préférable de pondérer les caractéristiques pour chaque phonème indépendamment.*

Hypothèse 10.2.2 (Pré-sélection par diphone). *Le diphone est une unité qui condense suffisamment d'informations segmentales pour autoriser la suppression de toutes les informations segmentales de la pré-sélection. La structure du diphone est en outre l'occasion d'exprimer de manière concise de nombreux critères suprasegmentaux.*

Hypothèse 10.2.3 (Durée et pause). *La durée syllabique est avant tout une conséquence du caractère éminemment mécanique de la respiration. Parmi les nombreuses caractéristiques de la respiration, la pause est particulièrement discriminante. En français, la syllabe qui précède directement une pause est toujours allongée, l'importance de son allongement dépendant de la structure interne de la syllabe.*

2.3 Principes d'optimisation

Hypothèse 10.2.1 (Sélection par FSMs). *Les machines à états finis constituent un mode de représentation idéal pour optimiser le processus de sélection d'unités non uniformes. Ce mode de représentation permet de précalculer la pré-sélection, le coût cible et le coût de concaténation tout en respectant l'algorithme de sélection initial.*

Postulat 10.2.1 (Contraintes de concaténation). *La structure de représentation ne peut modifier l'algorithme de sélection de manière involontaire. Elle peut par contre être l'occasion d'ajouter des contraintes supplémentaires qui améliorent l'algorithme.*

Hypothèse 10.2.4 (Contraintes de concaténation). *Une machine à états finis pondérée permet de respecter l'algorithme de sélection, tout en refusant certaines transitions nuisibles à la qualité globale de la sélection.*

3 Correction orthographique

3.1 Architecture du système

Postulat 15.1.2 (Approche globale). *La correction orthographique doit faire partie intégrante du processus d'analyse linguistique, de manière à profiter au maximum de l'ensemble des informations disponibles. Le processus de correction peut donc proposer des solutions en cours de traitement, mais doit attendre la fin du traitement avant de prendre une décision.*

Postulat 15.1.3 (Gestion des treillis). *Etant donné une erreur, les solutions trouvées par une étape de correction peuvent en proposer un découpage différent. Il devient dès lors difficile de les mémoriser dans la structure de données du synthétiseur, dans l'attente d'un choix ultérieur.*

Hypothèse 15.2.1 (Gestion des treillis). *Les machines à états finis permettent de conserver les solutions d'un traitement sous la forme d'un treillis où les divergences de découpage importent peu.*

Postulat 15.1.4 (Limitation des transferts de données). *Tout traitement doit éviter de stocker dans la structure de données des informations qui seront modifiées ou supprimées par les traitements suivants. Respecter cette contrainte limite le temps imparti à la conversion des données entre traitement et structure de stockage.*

Hypothèse 15.2.2 (Limitation des transferts de données). *Les machines à états finis évitent les transferts inutiles entre le traitement et le stockage, en conservant l'ensemble des informations dans un format directement utilisable d'un traitement à l'autre.*

3.2 Analyse morphologique

Hypothèse 15.5.1 (Dictionnaire d'IVs). *En synthèse, les formes lexicales connues du système ne nécessitent pas d'analyse morphologique. Un dictionnaire suffit amplement, pour autant que la structure représentant le dictionnaire soit légère et autorise un accès rapide. Les machines à états finis respectent ces contraintes.*

Hypothèse 15.5.2 (Analyse morphologique des OOVs). *En synthèse, les formes lexicales inconnues sont les seules formes à nécessiter une véritable analyse morphologique. Cette analyse est nécessaire afin d'orienter au mieux le choix de la catégorie à attribuer à la forme lors de l'analyse syntaxique.*

3.3 Correction

Postulat 15.1.1 (Erreurs audibles). *En synthèse, un système de correction doit se concentrer sur les erreurs d'orthographe qui sont décelables dans le flux de la parole, parce qu'elles en rendent l'écoute inconfortable.*

Hypothèse 15.2.3 (Approche stratifiée). *La correction orthographique peut utilement s'organiser en couches, de manière à corriger progressivement les formes qui contiennent différents types d'erreurs.*

Hypothèse 15.2.4 (Casse et accentuation). *Il est fréquent qu'une forme lexicale ne soit pas reconnue par l'analyse alors qu'elle ne diffère de la forme recensée dans le lexique qu'au niveau de la casse ou de l'accentuation. Il est dès lors important de détecter ces formes au plus tôt, afin de leur éviter un traitement lourd à réserver aux véritables erreurs typographiques.*

Postulat 15.6.1 (Limites d'édition). *Le nombre d'erreurs autorisées dans une forme lexicale doit tenir compte de trois facteurs : la longueur de la forme, le type de l'erreur et les limites de la compréhension humaine.*

Hypothèse 15.6.1 (Limites d'édition). *Un filtre de composition est le média idéal pour modéliser aisément les trois facteurs qui influent sur le nombre d'erreurs présentes dans une forme lexicale.*

Hypothèse 15.6.2 (Modèle de langue flexionnel). *Les performances des modèles de langue en analyse syntaxique laissent penser qu'un modèle de langue, combinant les catégories syntaxiques et les traits grammaticaux, peut efficacement gérer les erreurs d'accord présentes dans une phrase.*

Postulat 16.2.1 (Erreurs $n \rightarrow m$). *Une gestion des erreurs $n \rightarrow m$, indépendante du contexte, est une nécessité en reconnaissance des caractères.*

Hypothèse 16.4.1 (Erreurs $n \to m$). *Les machines à états finis permettent l'expression aisée de règles de gestion des erreurs $n \to m$ indépendantes du contexte.*